The Merlin Process Transactions, specified with Socca

Master’s thesis of Joris Höppener

supervised by S. Sachweh & L. Groenewegen

Preface

The first half year of 1994, I've been in Dortmund. In this half year, I've made the main part of this thesis at the University of Dortmund. Constructing such a thesis costs of course a lot of time. Constructing a good thesis also requires good supervisors. I was lucky. In Dortmund, Sabine Sachweh supervised me. Because of my domicile in Dortmund, Sabine has helped me a lot. The many and fruitful discussions with Sabine and her perfectionism realy inspired me.

Darum will ich Sabine herzlich danken, daß sie mir betreuen wollte und für ihre Zeit, die sie frei gemacht hat. Ich bin wirklich sehr froh, daß Du mir betreuen wolltest.
Natürlich will ich auch Prof. Schäfer danken, für die Möglichkeit um ein halbes Jahr in Dortmund an die Universität zu arbeiten. Es hat mir sehr gefallen um ein halbes Jahr ein Teil zu sein von der Lehrstuhl.

In Leiden, I had also a supervisor. This supervisor, Luuk Groenewegen, helped me to understand Socca. Before I went to Dortmund, he has taught me for half a year the ins and outs in Socca and PARADIGM. Furthermore, I really appreciated it, that Luuk went to Dortmund, so he could hear my lecture over my work.

Hartelijk bedankt Luuk. Je enthousiasme over het werk was voor mij erg aanstekelijk en de discussies waren erg leerzaam en gezellig.

Furthermore, my professor in Leiden, Prof. Engels, has helped me to go to Dortmund. I have seen him a couple of times in Dortmund. We then both talked in German, instead of Dutch. This made it easier for my german colleagues to understand where we talked about.

Besides making this thesis, I have also helped Gerald Junkermann in Dortmund, to make clear what the differences are between Socca and Escape. To make this distinction more clear, I have written a paper for him. Escape is developed by Gerald and is used to describe the process model in Merlin with use of EER and state charts. Therefore, Socca and Escape have some resemblance.

Contents

1. General introduction

This master's thesis is mainly written at the University of Dortmund. In this thesis, concepts, developed at the University of Leiden and the University of Dortmund are used.
The used concepts of Dortmund are over the Merlin process transaction model. Merlin is a Process centred Software Development Environment. To control parallel access on the documents, which are created during the software development process, a process transaction model is used.
The specification formalism Socca, developed in Leiden is used to specify this transaction model.

This thesis has two goals.

First, Socca will be tested by using Socca to specify a very complex system. In this way, it is tried to find weaknesses and problems in Socca.

Second, it will be shown how systems can be described. Often systems are described in a textual way or with use of code. The process transaction model will be described with the specification language Socca. It will be find out whether a specification language like Socca fulfils all expectations like a high level of abstraction, easy to understand and a clear structure compared to plain text or code.

In the next chapters, first introductions into the concepts are given.
In chapter 2, a general introduction to Merlin is given. In chapter 3, the Merlin process transactions are discussed. The introduction to Socca will be given in chapter 4.

After the introductions, the specification follows in chapter 5.

In chapter 6 and 7, the results of chapter 5 are evaluated concerning the above two defined goals. In chapter 8, a general conclusion on the total thesis is given.
In chapter 5, not the complete PARADIGM part in Socca is discussed. In the appendix, the total PARADIGM part of the specification can be found.

2. Introduction to the PSDE Merlin

This section contains a very brief introduction to Merlin, which is a prototype of a Process-centred Software Development Environment (PSDE). A Process-centred Software Development Environment coordinates and supports development activities in a software process, according to an underlying process description. This prototype is developed by Dortmund University and STZ, a Dortmund based software house. Information concerning Merlin can be found in [JPSW 94], [PSW 92], [Wo 94].

A process description in the PSDE Merlin can be described by the following four entities:

- Activities: A collection of tasks, which achieve a goal (e.g. specify, edit, compile or test)

- Roles: Groups of activities, which are logically highly related (e.g. project manager, programmer, tester)

- Documents: Objects of any type, produced during the software development process (e.g. c-modules, test plans, specifications).

- Resources: Technical resources such as tools and people, who participate in the production of software (e.g. editors and debuggers).

For instance, a c-module (Document) is implemented (Activity) by a Programmer (Role), using an editor (Resource).

When an engineer starts Merlin, he gets two windows on his screen. These two windows are called the WorkBench window and the WorkingContext window.

The WorkBench window is the main window and appears after Merlin is started. In this window, one can set the project one wants to work for and the role in this project. Now, the corresponding WorkingContext window can be requested.
In the WorkingContext window, all documents, which are needed by an engineer to perform a particular role, are displayed. The activities on the documents can be selected with context sensitive menus. The relations between the documents are presented by labelled arrows. Furthermore, the engineer can set the tool he wants to use for a certain activity.
In a project, an engineer may have several roles. This means that for each role in a specific project, the engineer gets another working context.

In figure 1, the Merlin interface of an engineer, called Miller, with the role programmer and the project merlin_demo_project is presented. All the buttons and menus of the windows in figure 1 will now be explained.

The WorkBench window has four buttons. With the Accounting button, the engineer can login or logout. The Project and the Role button are needed to choose the project respectively the role. If the entry working context is chosen from the menu of the button Tool, the WorkingContext window appears.
In the WorkingContext window of Miller, four accessible documents are displayed. Next to each document, the activities, which Miller can start on the documents, can be chosen with

context sensitive menus. The documents don’t have the same type. Two of them are c-modules and two of them are specifications. In the example, Miller is a programmer.

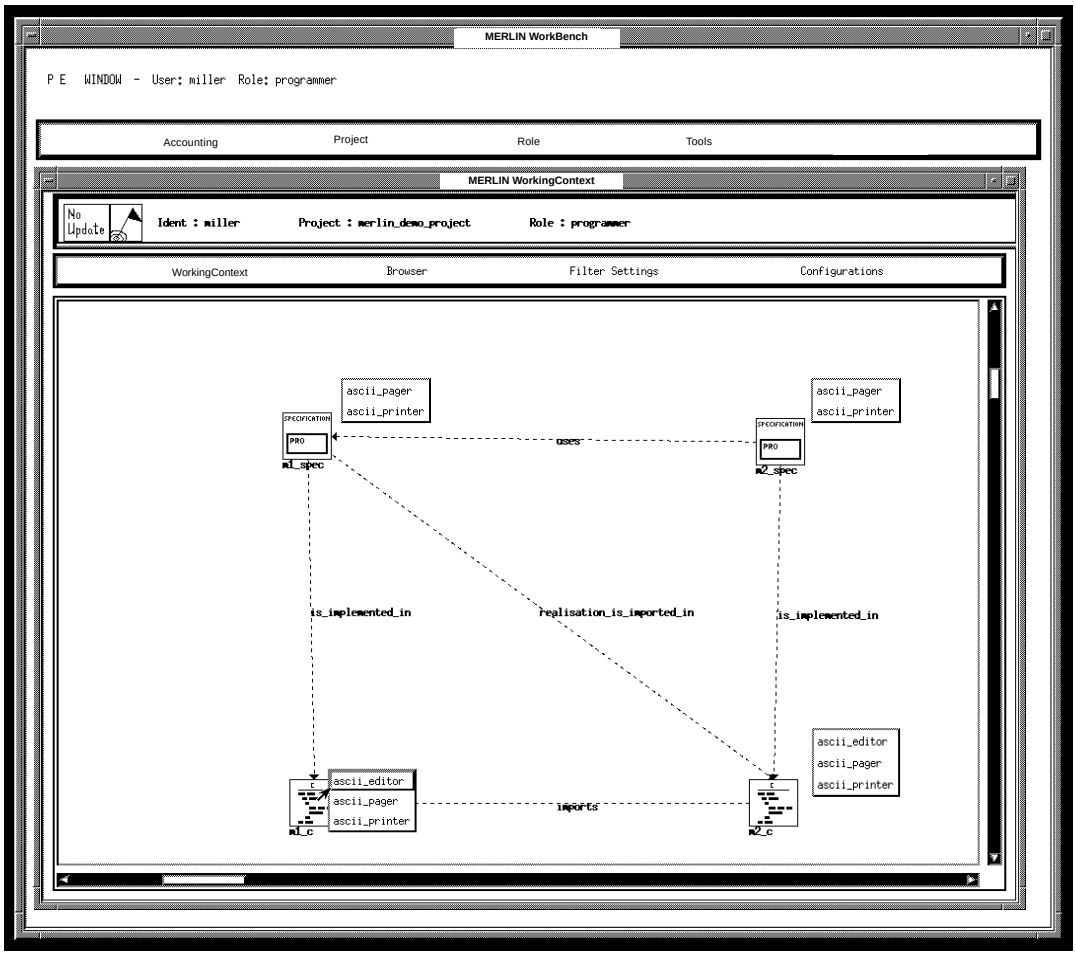


FIGURE 2.1. Interface of the PSDE Merlin.

The engineer can set the kind of tool, he wants to use for a specific activity, with the Configurations button.
With the Filter Settings button, the engineer can filter some documents or documents types out of his working context. In this way, the engineer isn’t bothered with momentary unnecessary information.
The WorkingContext button is used to shut down the working context and the Browser button is needed to structure the document graph in the working context in a different way.
When an activity is performed and its result influences the working context of an engineer, this engineer is informed about the changes by the update flag in the top left corner. Then, the engineer can choose whether he wants his working context to be updated.

A document consists of the contents and the status. The contents is the text or binary representation and the status shows where the document is in its life cycle. When a certain mile stone is reached, the status of the document is changed by the process engine or the engineer.
As already mentioned, an engineer only has a document in his working context, if he needs this document to perform an activity. When the engineer has finished a document, or when the engineer can’t work on a document, before other activities are finished, it is irrelevant for the engineer to see this document in his working context. Whether a document is relevant for an

engineer or not depends on the status of the document and on the underlying process description.
The next example makes the usage of document status in Merlin more clear. In figure 2 and 3, not the total screen dump is given. Only the parts of the screen, in which we are interested, are shown.

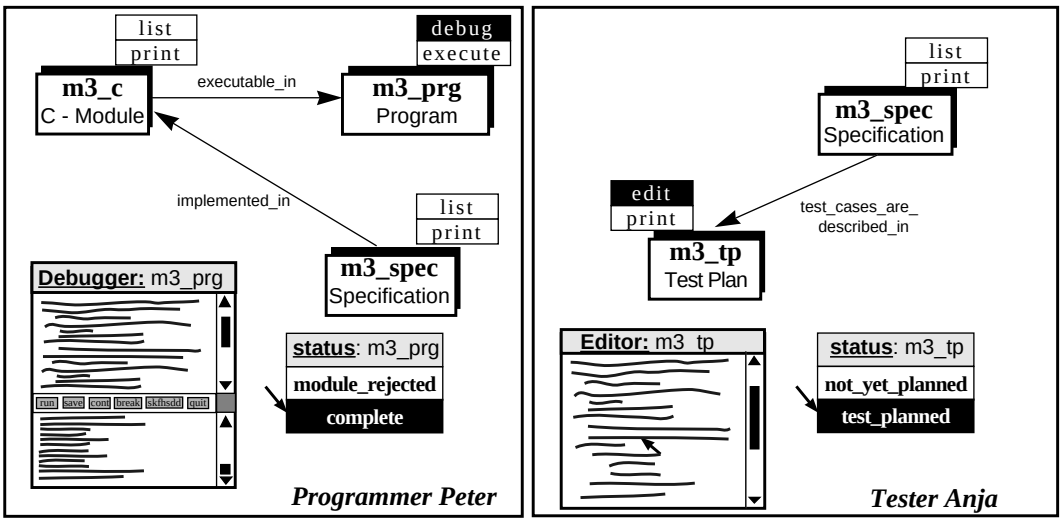


FIGURE 2.2. Cooperation between a programmer and a tester.

The programmer Peter has three documents in his working context. Peter needs the specification to implement the c-module. The activities on the specification are list and print. So, Peter can only read the specification. Peter has already completed the c-module and is now using the debugger, to check whether the implementation is correct.
Anja will test the c module, which is just completed by Peter. To test the c module, Anja has made a test plan. For that goal, she has read access on the specification. In this test plan, the test strategy and the test cases are described.
On this moment, Peter has completed debugging successfully and Anja has created her test plan. The status of the documents m3_prg and m3_tp are changed. The two working contexts of Anja and Peter can be updated. In figure 3, the updated working contexts are presented.

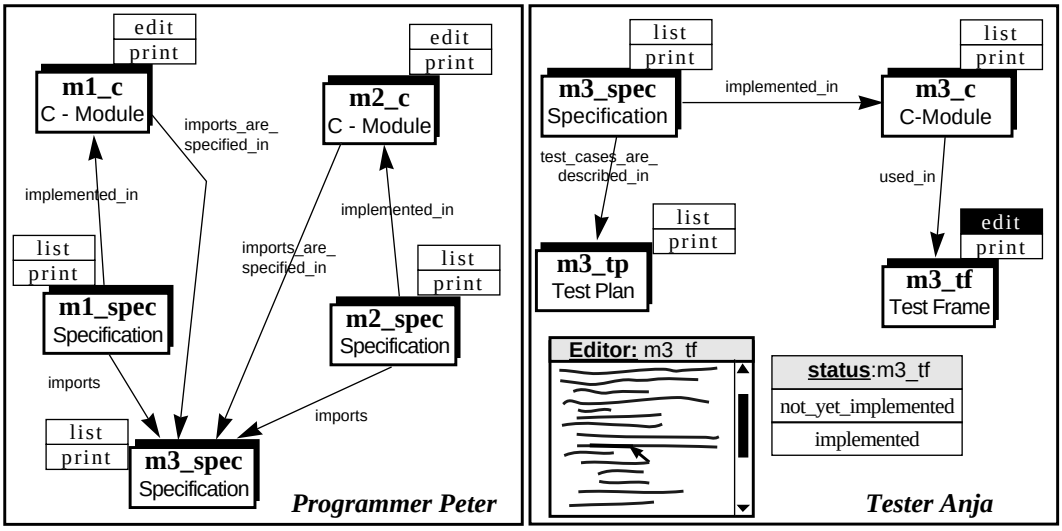


FIGURE 2.3. Updated working contexts.

Peter can now implement the c modules m1_c and m2_c. Only after successful implementation of the m3_c module, the m1_c and m2_c module can be implemented, because the m1_c and the m2_c module import the m3_c module. If Anja would reject the m3_c module, it would be given back to Peter. The documents m1_c and m2_c would disappear out of the working context of Peter, because m3_c has not been implemented successfully. To implement both c modules, Peter can read three specifications.
Anja writes a test frame. With use of this frame, a test program has to be created. Anja creates m3_tf, with use of the already created test plan and the specification. Anja can't write in the test plan any more. Eventually, Anja has finished editing and can now change the status of the document m3_tf.

The process engine makes sure that the working contexts of the engineers are updated. Besides this, the process engine also starts activities on documents, that needs no user interaction. For instance, the process engine automatically compiles a c module, when a programmer just implemented a c module successfully.

As generally observed, a lot of cooperative work exists in a process-centred Software Development Environment. To coordinate the parallel access on documents in Merlin, a process transaction model was defined, which is described in the following chapter.

3. Introduction to the process transactions

In the previous chapter a short introduction of Merlin, a Process centered Software Development Environment (PSDE), is given. Such a PSDE supports the coordination of all the activities, executed by engineers and the process engine. Without coordination, two or more engineers could simultaneously work on the same document, without knowing from each other. This would very fast lead to inconsistencies. To make sure that all the activities on documents are coordinated, process transactions are used.
This chapter is a summing-up of the dissertation of Stefan Wolf [WO 94]. The biggest part of this chapter can be found back in chapter 6 of this dissertation.

The process engine initiates process transactions. It either initiates them, because an activity of the engineer needs a process transaction or the process engine initiates a transaction as a reaction on the transaction of the engineer.
When an engineer changes the status of a document, this can have impact on other documents and their status. If for instance the status of a specification module has been set to *Complete* by an engineer, the process engine changes the status of the related C module from the value *Incomplete* back to *Not yet implemented*.

Whenever a transaction is initiated, the process engine or the engineer wants to have access on the contents and status of a document. To guarantee, that parallel access on documents won't lead to inconsistency of the documents, locks and time stamps are used by the transaction model.
Pessimistic transactions can ask for locks on contents and/or status of a document. These locks restrict the access on this contents or status for other transactions. If a transaction has obtained a write lock, no other transaction may read or write on this contents or status. Granting such a request would be incompatible. If a transaction holds a read lock, other transactions may only read this specific data, but can't change the data. If a transaction requests a lock, but it can't obtain the lock immediately, because other transaction(s) already hold this lock and granting the request is not compatible, there is a conflict. Either the transaction, which requested the transaction, or the transaction(s), which hold the lock have to be aborted.
Optimistic transactions can ask for time stamps on contents and status of a document. Time stamps don't restrict the access on the documents. At the end of execution, a transaction (let's call it now T1), which uses time stamps, will check whether other transactions used its documents. If the activities of these transactions are in conflict with the activities of T1 on this document, the transaction T1 will be aborted and the changes of the document, made by transaction T1, aren't stored.
Transaction T1 uses a log, to check whether other transactions used its documents. In the log, all the information of released locks and already used time stamps is stored. With this information, it is checked whether time stamps of transaction T1 are in conflict. It is also checked whether the time stamps of transaction T1 are in conflict with transaction, which now have locks.
Checking of time stamps is called validating.

The engineers and the process engine use tools to work on the documents. An engineer uses interactive tools and the process engine uses batch tools.

There are five types of transactions. Three types of transactions are initiated, because the engineer starts an activity. Two types of transactions are initiated, because they react on changes made by the activities of the engineers.
If the engineer opens its working context, the process engine checks whether this context has to be under pessimistic protection. If this is the case, a Pess_AF transaction will be started. Otherwise, the engineer can choose whether he wants pessimistic protection on whole its working context or only (optimistic or pessimistic) protection on a document of its working context, if he is using this document.

First the three types of transactions, which are initiated when an activity is started, are given.

Pess_Akt

This transaction is initiated, when the engineer wants an interactive tool with pessimistic protection to work on the document. The working context is in this case not build under pessimistic protection.
The transaction of this type obtains locks on one contents and status of the document, it wants to work on.
After the locks are obtained, the transaction protects that the engineer can work on the document safely. After the engineer is finished, the transaction will commit. This means that the locks are released and that information about this releasing is inserted in the log. After that, the transaction will be deleted.

Pess_AF

This transaction is initiated, when the engineer starts the working context and either the process engine or the engineer want that the working context has pessimistic protection.
The transaction of this type obtains locks on more then one document. The complete working context of the engineer will be locked. The working context of an engineer consists of all documents, where on the engineer has access with a specific role.
When the engineer is working on its documents and some information of the working context is changed, the engineer can refresh the working context. Refreshing means, that the locks of documents, which are no longer part of the working context, will be released and that locks on documents, which have become part of the working context, are requested. Also all the changed contents and status values of the documents are saved.
At the end, when the engineer leaves the working context, the transaction is committed and the log is updated.

Opt_Akt

This transaction is initiated, when the engineer wants an interactive tool with optimistic protection. The working context is of course not build with pessimistic protection.
A transaction of this type uses time stamps, instead of locks, to access contents and status. Only one contents and status of a document will be stamped.
At the end, the time stamps are validated. This means, it is checked whether other transactions had conflicting parallel access on the document of this transaction. If the validation is not

successful, the transaction is aborted. Otherwise, the time stamps are changed in locks and the type of the transaction is changed in Pess_Akt. The purpose of changing the type in Pess_Akt will be explained right away.

Now the two types of transaction, initiated to react on changes made by activities of the engineers, are described.
Both Auto and Kons transactions can be started, when an engineer has changed the status of a document. When an interactive tool has been used, the status of this document can be changed. The interactive tool had one of the three above described sorts of protection (Pess_Akt, Pess_AF, Opt_Akt).

Kons

A transaction can request as many locks, as it needs. So, a transaction may lock only a status of a document and not the contents itself.
After the locks are obtained, the transaction can use batch tools. At the end, the transaction is committed.

Auto

The Auto transaction behaves quite similar to the Kons transaction. It can also ask for as many locks it wants and use a batch tool. The difference between these two types is the priority of the type.

Both Auto and Kons are pessimistic transactions.
A transaction of the type Kons makes sure that the state of the project will stay consistent. The consistency of the project has to be guaranteed, therefore it must be avoided that the execution of a Kons transaction is delayed. The execution of a Kons transaction is delayed, when it can't obtain some locks, because another transaction has these locks.
A transaction of the type Auto performs some changes in the project state, which are less important. Therefore, it is not a real problem, when the execution of a Auto transaction is delayed for a short time.
Therefor, the type Kons has a higher priority then the type Auto because the higher the priority, the more likely it is that such a transaction wins the conflict over a lock.

Pessimistic engineer transactions are transactions of the type Pess_Akt or Pess_AF. This means that such a pessimistic transaction uses locks to restrict the access on documents or status.
Optimistic transactions are of the type Opt_Akt. Such an optimistic transactions doesn't restrict the access on the contents and status of the document, it was started on.

As already told, Auto and Kons transactions, are initiated as a reaction on engineer transactions. This results in nested transactions in the Process Transaction Model.
Transactions of the type Pess_Akt, Pess_AF and Opt_Akt are parent transactions. Transactions of the type Kons and Auto are child transactions.

Kons and Auto transactions propagate changes, which are done in an engineer transaction. Their is no limitation on the order and number of transactions of the type Auto and Kons, which follow after a specific father transaction (of the type Pess_Akt, Pess_AF and Opt_Akt).
The child transaction inherits the locks from its parent. It is only possible that two write locks exist on a contents or status, when a child inherits the write lock of its father.

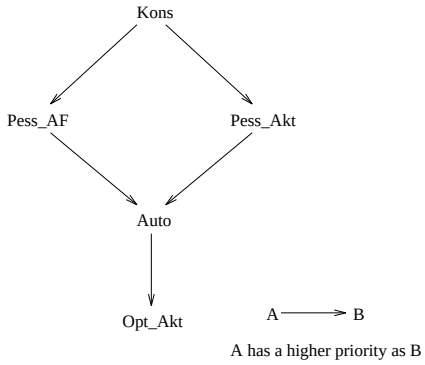
It is of course possible, that a transaction requests a lock, when there is already a transaction, which holds this lock. If the request is not compatible, then there is a conflict.
To solve this and also other kinds of conflicts, which are caused by parallel access, synchronization rules are defined.

Synchronization rules

Rule 1
When an engineer-initiated transaction, of the type Opt_Akt, Pess_Akt or Pess_AF, has write access on a document, it must have (read or write) access on the status of this document.

Rule 2
When an Opt_Akt transaction starts child transactions, the Opt_Akt transaction has to be validated successfully and it has to be converted in a pessimistic transaction of the type Pess_Akt. The time stamps on the documents and status are changed in locks.

Rule 3
Every transaction type has a priority. The order between priority of the transaction types is the following:



Rule 4
When a conflict occurs between two Pess_AF, two Pess_Akt or a Pess_AF and Pess_Akt transaction over a lock on a document or its status, the transaction, which requests the lock, will be aborted.

Rule 5
If a conflict occurs between two Kons transactions over a lock on a document or its status, the transaction, which requests the lock, will be aborted.

Rule 6
If a conflict occurs between a Kons transaction and a Pess_Akt or Pess_AF transaction over a lock on a document, the Pess_Akt or Pess_AF transaction, when this transaction has not started any child transactions yet, will be aborted.

Rule 7
If a conflict occurs between a Kons transaction and a Pess_Akt or Pess_AF transaction over a lock on a document, the Pess_Akt or Pess_AF transaction, when this transaction has started child transactions, will release this lock.

Rule 8
If a conflict occurs between a Kons transaction and a Pess_Akt or Pess_AF transaction over a lock on a status, the Pess_Akt or Pess_AF transaction will release the lock.

Rule 9
If a conflict occurs between an Opt_Akt transaction and an other transaction of a certain type over a lock or time stamp on a document or its status, the Opt_Akt transaction will be aborted.

Rule 10
If a conflict occurs between an Auto transaction and a pessimistic transaction over a lock on a document or status, the auto transaction will be aborted.

Rule 11
A nested transaction will end normally, when child transactions are aborted.

	pess_Akt (no Subtr.)	**pess_Akt** (with Subtr.)	**pess_AF** (no Subtr.)	**pess_AF** (with Subtr.)	**opt_Akt**	**kons**	**auto**
pess_Akt	SR 4 √ / Abort	SR 4 √ / Abort	SR 4 √ / Abort	SR 4 √ / Abort	Abort / SR 9 √	SR 6,8 √ / Abort	Abort / SR 10 √
pess_AF	SR 4 √ / Abort	SR 4 √ / Abort	SR 4 √ / Abort	SR 4 √ / Abort	Abort / SR 9 √	SR 6,8 √ / Abort	Abort / SR 10 √
opt_Akt	SR 9 √ / Abort	SR 9 √ / Abort	SR 9 √ / Abort	SR 9 √ / Abort	SR 9 √ / Abort	SR 9 √ / Abort	SR 9 √ / Abort
kons	Lock released / Abort / SR 6,8 √	Lock released / SR 7,8 √	Lock released / Abort / SR 6,8 √	Lock released / SR 7,8 √	Abort / SR 9 √	SR 5 √ / Abort	Abort / SR 10 √
auto	SR 10 √ / Abort	SR 10 √ / Abort	SR 10 √ / Abort	SR 10 √ / Abort	Abort / SR 9 √	SR 10 √ / Abort	SR 10 √ / Abort

In the above matrix, the synchronization rules are presented in another way. In each matrix field, it is presented how transactions will react if a conflict occurs and which synchronization rule prescribes this reaction.
A field in the above matrix has to be interpreted in the following way. A transaction Ti, which is requesting a lock or validating, with a type in the left column, is in conflict with a transaction Tj of the type in the top row. The left part of a matrix field gives the solution for transaction Ti, the right part the solution for transaction Tj.

The transaction concepts, described in this chapter, are specified in chapter 5. The formalism, which is used to describe this process transaction model, is called Socca. Socca will be described in the following chapter.

4. Introduction to Socca

The specification formalism Socca is developed at Leiden University.
The idea behind Socca is the separation of concern, which finds expression in the combination of the best fitting (parts of) different formalisms.
Socca uses three perspectives to define a specification, namely data perspective, behaviour perspective and process perspective. The process perspective is still in development, which has as consequence that this perspective won't come back in this master thesis. The process perspective should describe the data flow.
In the following section, the components of Socca will be briefly described. First the data perspective is defined. For further information over Socca, one could read [EG 93] and in [Gr 91] more information concerning PARADIGM can be found.

Data perspective

In the data perspective, the static structure of the system and its relation to its environment are described, by an object oriented approach with use of EER.

Besides the influence of sub-systems on other sub-systems, it will also be examined, what influences the system and what is influenced by the system. The major components (classes) of the system and the environment are modelled with use of EER.

To specify the data perspective, all the classes have to be specified and for each class, the export operations and attributes are given.

Furthermore, with four different sorts of relations, the dependencies between the classes are modelled. These relations are: Inheritance, Composed-of, Uses and General relations.
These four relations result in three diagrams. These are: Class hierarchy (including inheritance and composed-of), General relations between the classes and the Uses relations between the classes.

After the data perspective, the behaviour perspective is defined in the Socca specification.

Behaviour perspective

The behaviour perspective concentrates on the operations, which realize the transformation on the data. Especially the allowed sequence of operations stands central in this observation.
The behaviour perspective can be divided in three parts, namely external behaviour, internal behaviour and PARADIGM.

External behaviour.

In this part, the behaviour of a class, which is visible from outside, is specified. To specify the external behaviour of a class, the allowed sequences for calling the exported operations are

defined by means of State Transition Diagrams (STD's).

A State Transition Diagram exists of states, which are connected with arrows (transitions). If the behaviour processes the transitions, the state of the behaviour will be changed.
In this paper, a state chart feature is used to extend the STD's.
Sometimes, a STD has a couple of states, which have the same transition to a specific state X. Until now, all the transitions were given by there own arrow, in a Socca specification. This makes a STD sometimes difficult to survey.
With this next feature, a polygon is drawn around all those states, which have the same transition to state X. Now, only one arrow is drawn for this transition. This arrow starts at the polygon and ends in state X.
In figure 1, an example is given.

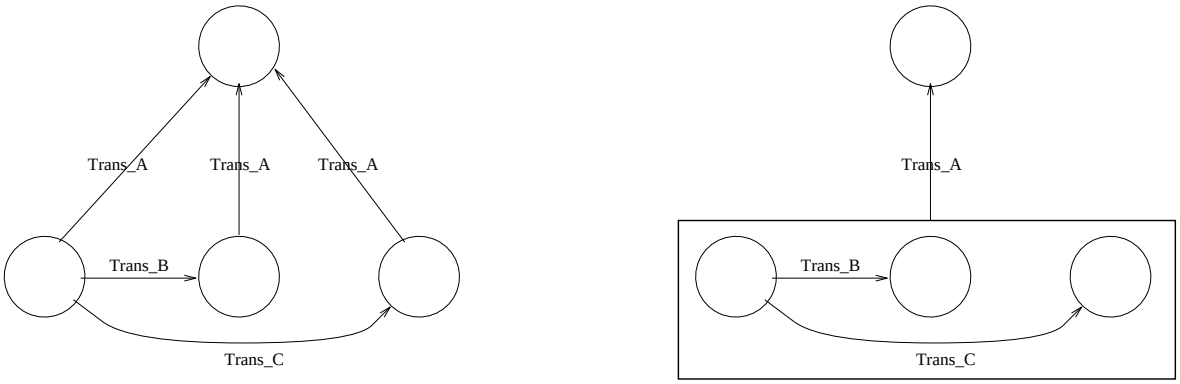


FIGURE 4.1. The same STD's, with and without the feature.

In figure 2, a small example of an external behaviour is given.

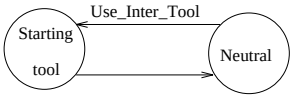


FIGURE 4.2. External behaviour.

In this external behaviour, a class has one export operation Use_Inter_Tool. By calling this export operation, the behaviour goes from state *Neutral* to state *Starting Tool*. The tool is started and the behaviour can stay in state *Starting Tool* or go to state *Neutral* and ask a new tool.
Next the internal behaviour is presented.

Internal behaviour.

The behaviour of each export operation is described. The export operation calls other export operations and internal operations of its own class. So, the internal behaviour specifies the allowed sequence of internal operations and of calling of export operations. To specify this sequence, again State Transition Diagrams (STD's) are used.
The calling of another export operation is shown in the STD by the prefix "call".
The internal behaviour of an operation is activated, when this operation is called by an export

operation. The activation is shown in a STD by the prefix “act”.
In figure 3 and 4, two small examples of internal behaviours are given.

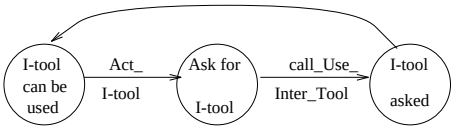


FIGURE 4.3. int-Ask_Tool: STD of the internal behaviour.

This internal behaviour describes the request for an interactive tool.
The transition ‘Act_I-tool’ activates this internal behaviour. After activation, an export operation is called. The name of this export operation is Use_Inter_Tool. This export operation makes sure that an interactive tool is started.

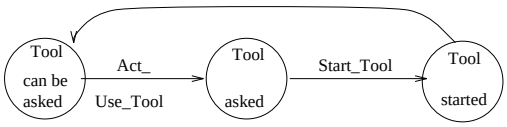


FIGURE 4.4. int-Use_Inter_Tool: STD of the internal behaviour.

This internal behaviour describes that the interactive tool is started.
The internal operation Start_Tool actually starts the tool.

Next, the third and last part of the behaviour perspective is presented.

PARADIGM.

PARADIGM is developed at the Leiden University by L. Groenewegen [Gr 91]. PARADIGM stands for PARallelism, its Analysis, Design and Implementation by a General Method. PARADIGM can be used to manage several to each other related (parallel) processes.

In this part, the coordination of the communication is specified. There are two sorts of coordination, which are modelled:

- Coordination of all internal behaviours of a class X.
This is the coordination between an external behaviour X and the internal behaviour belonging to the same object.

- Coordination of all internal behaviours, which call export operations of a class X.
This is the coordination between any internal behaviour of an operation, calling export operations of class X, and the external behaviour X.

This coordination is managed by the external behaviour of class X.
All the internal behaviours are called employees. The external behaviour is called manager.

To describe the coordination, several subprocesses and traps are defined for each employee. A

subprocess is a part of an internal behaviour. Some states and transitions of the internal behaviour are left out in the subprocess. In this way, the internal behaviour is restricted. This restriction of this internal behaviour reflects in what state the external behaviour is.
Traps are (sets of) states in a subprocess, in which an employee is ready to switch from one subprocess to another. A trap hasn't got any outgoing transitions. When the behaviour is in a trap, it can only leave the trap by switching from the current subprocess to a next subprocess.

In PARADIGM, it is described:
- in which subprocess an employee has to be, depending on the state of the manager.
- which trap triggers a transition between two states of the external behaviour of the manager.
PARADIGM will be explained by the following example.
The above specified internal behaviours are used as employees. The manager is the above specified external behaviour.
As already mentioned, two sorts of employee exists. The first kind of employees are from the same class, as the manager. The second kind of employees, are those calling export operations of the class from the manager. In figure 5, the second kind of employee is presented. In figure 6, the first kind of employee is visualized.

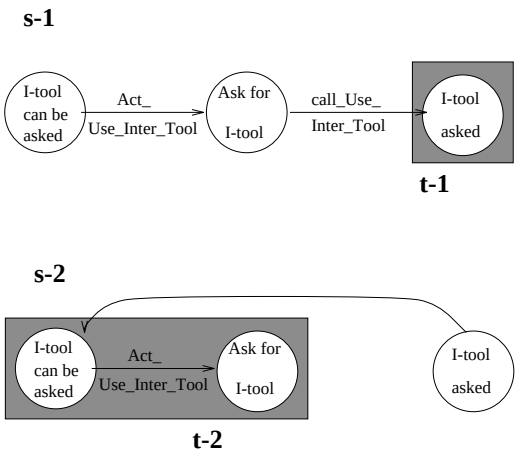
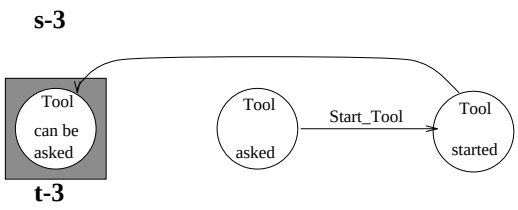


FIGURE 4.5. int-Ask_Tool's subprocesses and traps.

The two subprocesses (s-1 and s-2) are a part of the same internal behaviour. In subprocess s-1, the behaviour is busy with the operation. When it has called the operation Use_Inter_Tool, the employee is in trap t-1.
In subprocess s-2, the employee can't call the operation Use_Inter_Tool.
When the employee is in one of the traps (t-1 or t-2), it is ready to switch to the other subprocess.



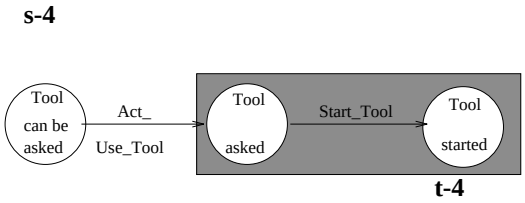


FIGURE 4.6. Use_Inter_Tool's subprocesses and traps.

In subprocess s-3, the internal behaviour can't be started. If the behaviour has reached trap t-3, it can switch to subprocess s-4.
In subprocess s-4, the internal behaviour can be activated. If the internal behaviour is in trap t-4, the behaviour has really started, i.e. the operation is busy starting the interactive tool.

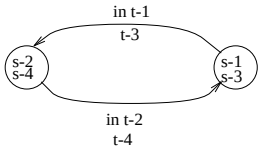


FIGURE 4.7. Manager of two employee.

The external behaviour of figure 1 is taken as a manager. In state *Neutral*, the two employees are in subprocess s-1 and s-3. If the external behaviour goes to state *Starting Tool*, the behaviour of the employees have to be in trap t-1 and t-3. When the employee is in trap t-1, the export operation has called the operation Use_Inter_Tool.
In external state *Starting Tool*, the employees have switched from subprocess. The internal behaviour of Use_Inter_Tool can now be activated.
If the external behaviour goes to state *Neutral*, both employees switch again of subprocess. The internal behaviour Use_Inter_Tool is activated and the operation Ask_Tool can call the export operation Use_Inter_Tool again.

This concludes the short introduction of the behaviour perspective and Socca. Next, the specification of the process transaction model with use of Socca will be presented.

5. Specification

In this chapter, the process transaction model of the PSDE Merlin of chapter 3 are specified with the specification language Socca, which has been briefly described in chapter 4. The first perspective of Socca is the data perspective.
It is assumed, that the reader understands the contents of the previous chapters. This will enable the reader to understand the following specification of the process transaction model easier.

5.1 Data Perspective

As already described in the previous chapter, Socca starts with the data perspective. The data perspective models the static structure of the components of the PSDE Merlin and more detailed the process transaction model. Description of such a static structure will be based on EER concepts. The contents of this section is described in a step-wise manner. There are roughly three steps.
First, complex structured objects are described by class hierarchies consisting of class definitions as well as of part-of and is-a relationships.
Second the missing general relationships are added.
Third, it will be indicated which of the classes indeed use the operations exported by other classes.

In figure 5.1.1, the class hierarchy is given. The PSDE Merlin consists of four subclasses.
The Process Description, which contains the information over the kind of software development process.
The Project Description, which contains the necessary information over the project, like all the documents, information over project members and tool software. A documents consists of the contents and a status.
The Process Engine controls the working context and work bench by executing the process description.
The fourth subclass is Cooperation Model, which enables the communication between the user and the process engine via the WorkBench Window and the WorkingContext Window. Furthermore, it coordinates all activities of all users by a process transaction concept
With use of the WorkBench Window, the engineer can choose the right project, where on he wants to work with a specific role and he can request for the WorkingContext Window.
The WorkingContext Window consists of all the documents, which can be used by an engineer.
The class Process Transaction Model makes sure that all the activities won't jeopardise the consistency of the data. Therefore, it coordinates all the access on documents. A transaction can be either optimistic or pessimistic. There are five types of process transactions distinguished: Opt_Akt, Pess_Akt, Pess_AF, Kons and Auto Transaction. The types Auto, Kons, Pess_Akt and Pess_AF are always pessimistic and the type Opt_Akt is optimistic.
To control access on contents and status, locks and time stamps are used. Pessimistic transactions only use locks and optimistic transactions only time stamps. When a transaction is suc-

cessfully ended, information about their locks or time stamps is stored in tuples, which built up the log. The log is used by the Opt_Akt transaction, to check at the end of its execution whether there was no parallel access on the document or status, it had worked on.

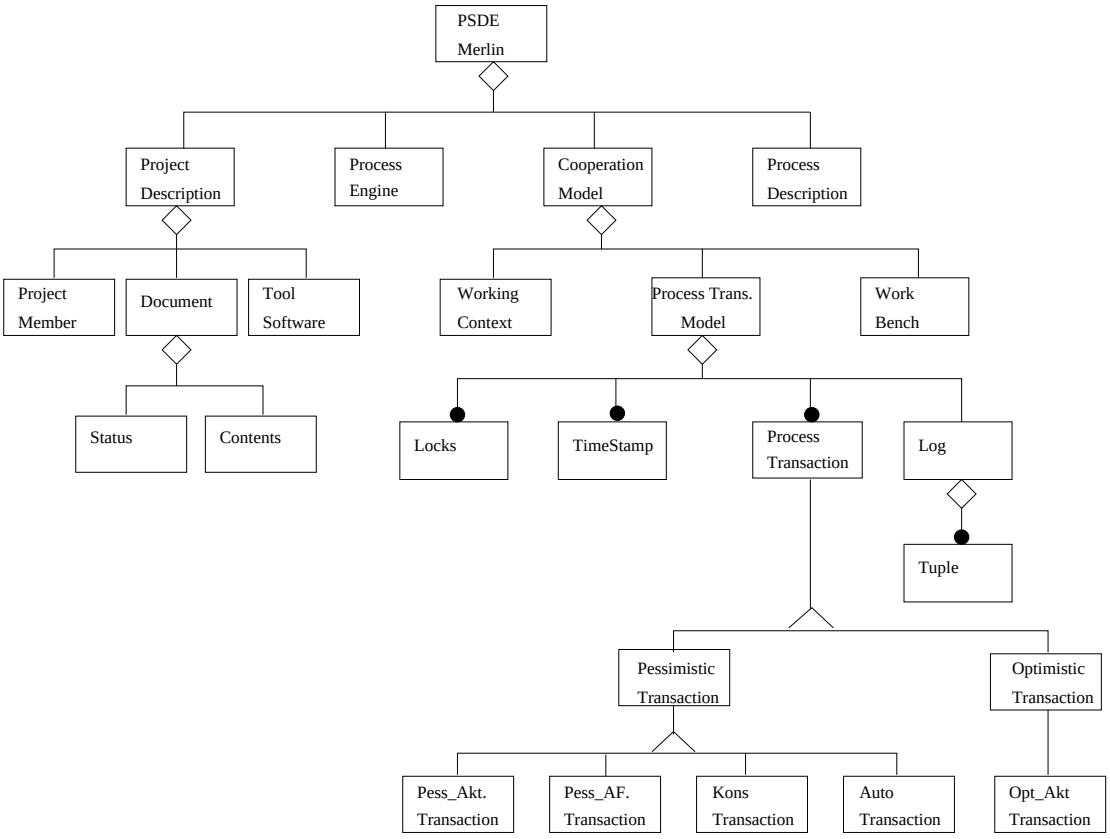


FIGURE 5.1.1. Class hierarchy.

The black dot indicates that the number of instances participating in a relationship may be between zero and n. The small triangles indicate is-a relationships and the small diamonds indicate part-of relationships.

Because this data perspective is used to give an overview of the Merlin structure, not all the classes will be used in the further specification, as this thesis is focused on the process transaction model.
Therefore, the classes PSDE Merlin, Project Description, Project Member, Tool Software, Process Description and Cooperation Model are not considered in the remaining specification, i.e. they are out of scope.

In the dissertation of Stefan Wolf [Wo 94], the internal organisation of the log is not clearly described. There is described, that some information is inserted in the log, after a lock or time stamp is released. The only information is given on page 152 of the dissertation, where it is shown, how in the Merlin implementation the log is filled with information of a released lock. As no term was given for an entry of the log, the term tuple is used in this thesis to denote one entry in the log. Insertion of such a tuple in the log can be compared with the *INSERT* operation on page 152 in the dissertation of Stefan Wolf.

The attributes and export operations of the relevant classes in figure 5.1.1 are given separately in figure 5.1.2, for the sake of readability.
At this moment, it is sufficient to have a general idea what the meaning of the export operations and attributes in figure 5.1.2 are. Later on in the behaviour perspective, these export operations will be described in more detail.

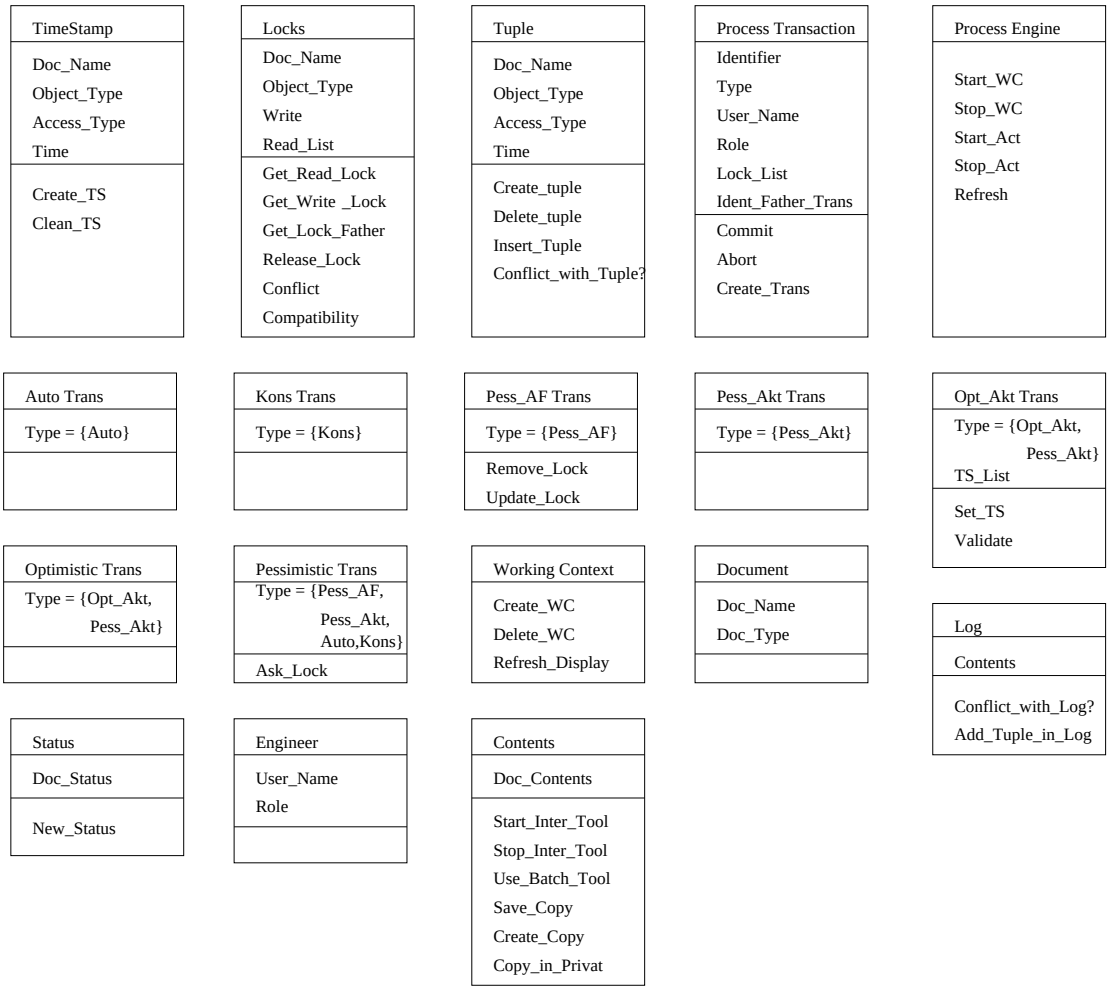


FIGURE 5.1.2. Attributes and operations.

At first the meaning of the important attributes will be explained.
The attribute Doc_Name represents the name of an document. Each document has an attribute Object_Type. The value of this type is either contents or status. The attribute Access means how the contents or status of a document is accessed. A time stamp or lock can only have read- or write access.
The attribute Lock_List contains all the locks, which have been set by a transaction. Analogously TS_List contains all the time stamps set by an Opt_Akt transaction. The attribute Type of class Opt_Akt Transaction can have both the values Opt_Akt and Pess_Akt. This is so, because after successful validation of an Opt_Akt transaction the value of the attribute Type will change to Pess_Akt.
Now some important operations will be explained.
The operation Ask_Lock requests a lock on a contents or on a status. A request doesn't mean that the lock will be obtained, but it can be obtained. Analogously, the operation Set_TS is

used to set a time stamp on a contents or status. After such an operation, it is certain that a time stamp is created. In the class Lock, there are three kinds of operations to get a lock. A read lock with Get_Read_Lock, a write lock with Get_Write_Lock and a child transaction obtains the lock of its father with Get_Lock_Father. Release_Lock is used to release a lock. In class TimeStamp, Create_TS is used to create a time stamp and Clean_TS to delete the time stamp.

The five classes Auto -,Kons -, Pess_AF -, Pess_Akt and Opt_Akt Transaction inherit all the operations and attributes from their parent class Process Transaction and from Pessimistic Transaction or Optimistic Transaction.

It would be wise to look at figure 5.1.2 once more, after the rest of the data perspective has been read. The meaning of some operations will then be perhaps more clear.

The next step is to describe the general relationships between the relevant classes in more detail.

The creation of a transaction has to be controlled. This is done by the Process Engine. The Process Engine can start a transaction as a reaction on a demand of the engineer. The engineer can express his demands to the process engine through the working context or the work bench. Via the work bench, the working context can be created and displayed. Because of the demand of the engineer, a transaction of the type Pess_AF, Pess_Akt or Opt_Akt is created. The process engine can also start transactions to keep the project consistent or to use automatically certain tools. These transactions are of the type Auto and Kons.

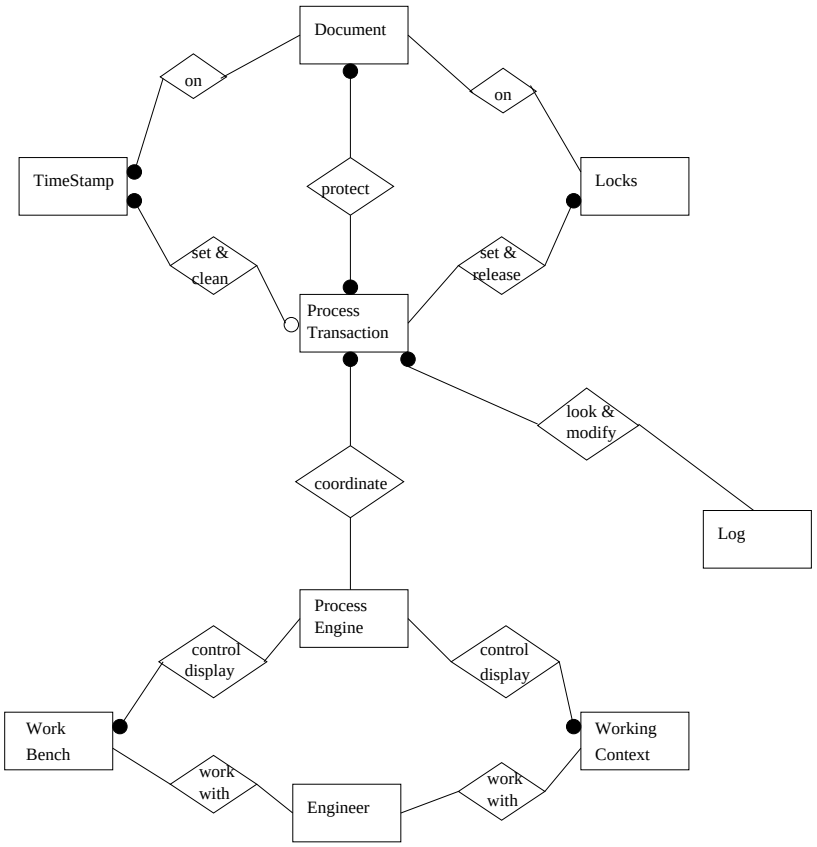


FIGURE 5.1.3. General relationships.

A transaction is always related to a name of an Engineer. This is the name of the Engineer, who directly or indirectly initiates a transaction. The indirectly initiated transactions are of the type Auto and Kons and are called child transactions. These child transactions are initiated when a parent transaction, which is initiated directly by the engineer, is stopped. The child transactions change the project situation, because of the impact of the execution of the parent transaction.
This Engineer has of course always a role. Therefore the name of the Engineer and his/her role are related to the transaction.

When a transaction is created, the transaction gets an unique identifier. A child transaction also gets an unique identifier. To use documents, the transaction asks for time stamps or requests locks. A lock on the contents or status will make sure that no other (incompatible) lock can lock this contents or status. So a lock restricts the access on a document.
A time stamp doesn't restrict the access. An Opt_Akt Transaction checks at the end of its execution with use of the time stamps, whether there was a conflicting parallel access on the contents or status of the document, it worked on.
The time stamps are cleaned (deleted), when the optimistic operation is validated. The type of the transaction will be changed in Pess_Akt, after the validation was successful. Now, the transaction can initiate child transactions. If the validation was unsuccessful, the Opt_Akt Transaction will be aborted.

When a transaction is requesting a lock, it is checked whether this request won't give a conflict. A conflict does arise, when the requesting lock isn't compatible with the lock(s), which are already held on the contents or status. Which transaction(s) have to release a lock or have to be aborted depend partly on the type of the involving transactions. If a conflict doesn't arise, the lock request is granted and the lock is set.
The precise rules for synchronizing this conflict are summarized in the Process Transaction chapter.

In the Log, it is stored when which locks have been released by committed transactions and when which time stamps are cleaned by successfully validated transactions. When a lock is released or a time stamp is cleaned, a tuple will be created, containing the necessary information about this lock or time stamp. The tuple is inserted in the log, when the transaction is committing or is validated successfully. Aborted transactions and unsuccessfully validated Opt_Akt transactions don't insert information in the Log. When an Opt_Akt transaction is validating, it will look in the Log to check whether some other transaction, which has already released the locks or time stamps on this document or time stamp, is in conflict with this opt_Akt transaction. If there is a conflict, this Opt_Akt transaction must abort. The data used by this Opt_Akt transaction could be inconsistent.

The third and last step for describing the data perspective is to indicate which of the classes indeed use the operations exported by other classes. All the uses-relationships are graphically given in figure 5.1.4.

The classes Engineer and Work Bench are out of the scope of the model of this specification. This means, that it is interesting what the influence is of these classes on the model and which operations are imported by it, but it is not interesting, which export operations are offered by these two classes.

The classes Engineer and Work Bench have no export operations in this model. The engineer uses (Uses1) the Work Bench to tell the Process Engine what working context has to be shown. The engineer also uses (Uses2) the Working Context to trigger the start of interactive tools on documents.
The WorkBench uses (Uses3) the Process Engine to pass the information through from the engineer. The Process Engine uses (Uses4) the WorkBench, to update the display of the work bench window.
The WorkingContext uses (Uses5) the Process Engine to start interactive tools on documents. The Process Engine uses (Uses6) the WorkingContext to update the window.

The Process Engine uses the Status (Uses11) and the Contents (Uses12), to save a new status or to start and stop tools on the contents.
The class Process Transactions uses (Uses14) Contents to create and save copies of the contents in the data base.

The Process Engine initiates and controls the process transactions (Uses7).
The Uses8 and the Uses9 relation in figure 5.1.4 shows that the Process Engine uses other export operations, when it controls a pessimistic or optimistic transactions. There are also some operations of the class Pess_AF Transaction, which are used by the Process Engine. This is made clear through the Uses10 relation.

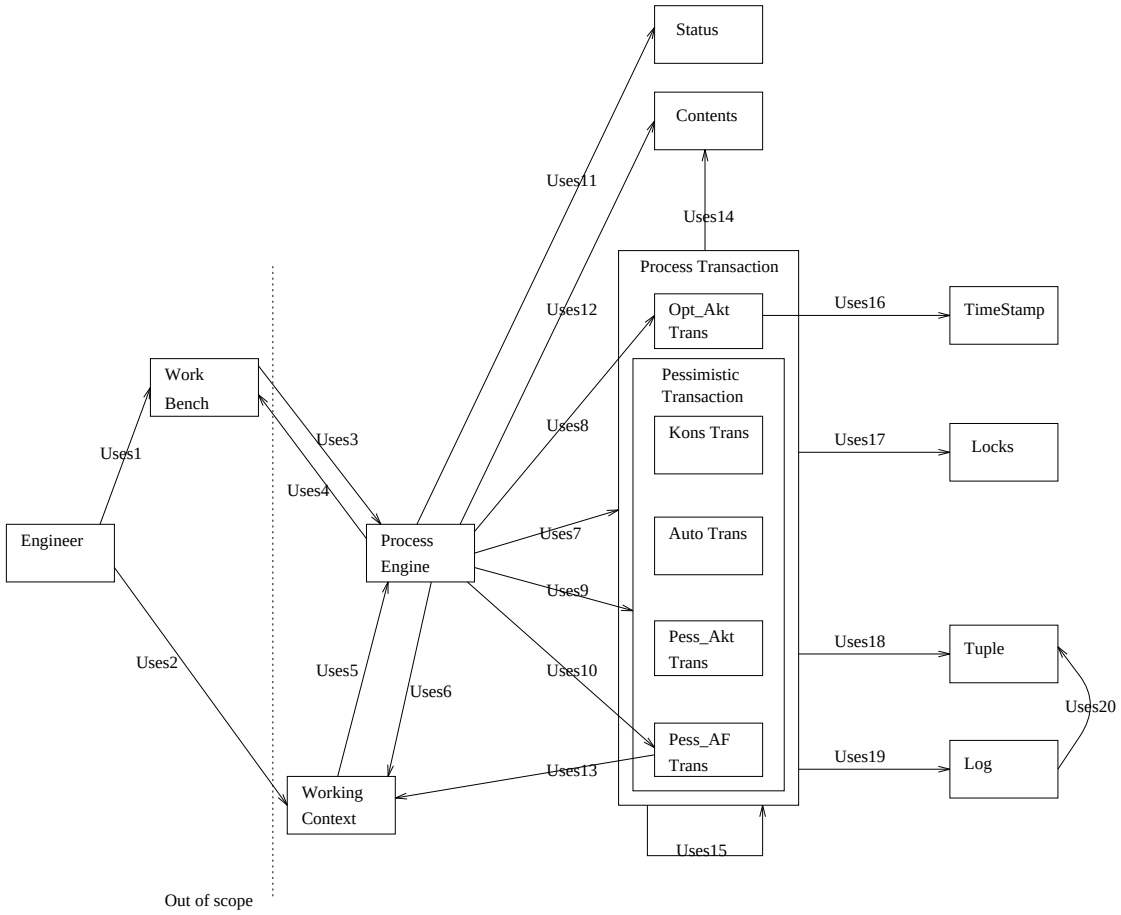


FIGURE 5.1.4. Import / export diagram.

The class Pess_AF Transaction only uses (Uses13) the Working Context, when the working context has to be deleted, because the transaction has to be aborted.

The Uses15 relation shows that one transaction can trigger transactions to abort.
A transaction can abort its self. This happens when the transaction requests a lock, which it can't obtain or when an Opt_Akt Transaction is validated unsuccessfully.
Or a transaction can abort other transactions, which were in conflict with this transaction and have to release a specific lock.

All the Process Transactions use (Uses17) the class Lock. Also the class Opt_Akt Transaction uses the class Lock, because after successful validation, the type of the transaction is changed in Pess_Akt and the transaction has changed its time stamps into locks.
All the Process Transactions also use (Uses18) class Tuple. The class Log inserts (Uses20) the tuples in its contents. Therefore, all the Transactions also use (Uses19) the class Log.

Only Opt_Akt Transaction uses (Uses16) the class Time Stamp.

This concludes the data perspective of the specification. Next, the behaviour perspective will be presented. The behaviour perspective will be started with the external behaviour.

5.2 External behaviour

The previous section has discussed the data perspective. The classes are defined, by giving there attributes and operations. How the behaviour of the classes is, has been globally indicated. This section will define the external behaviour of these classes. This external behaviour is visible from outside, specifying the allowed sequences for calling the exported operations. As already mentioned, some classes are only used in the data perspective, which means that no external behaviour will be specified for these classes. Furthermore, from the superclasses Document, Process Transaction, Optimistic Transaction and Pessimistic Transaction, the external behaviour is not given, because it is sufficient to give the external behaviour of their subclasses.
The behaviour of the exported operations is defined in the internal behaviour section. This internal behaviour comes afterwards. The internal behaviour specifies the allowed sequences of private operations and of calls of operations imported from elsewhere.
To describe the behaviour, state transition diagrams (STD) are used.

We will start with the description of the external behaviour of the subclasses of the class Process Transaction. In order to understand exactly what has to be specified, the export operations of the class Kons Transaction are listed down below. The first three operations are the same for all the Process Transaction subclasses, as they are inherited. The operation Ask_Lock is used by all the subclasses of Pessimistic Transaction.

Create_Trans
Commit
Abort
Ask_Lock

The external behaviour of the class Kons Transaction is shown in figure 5.2.1.

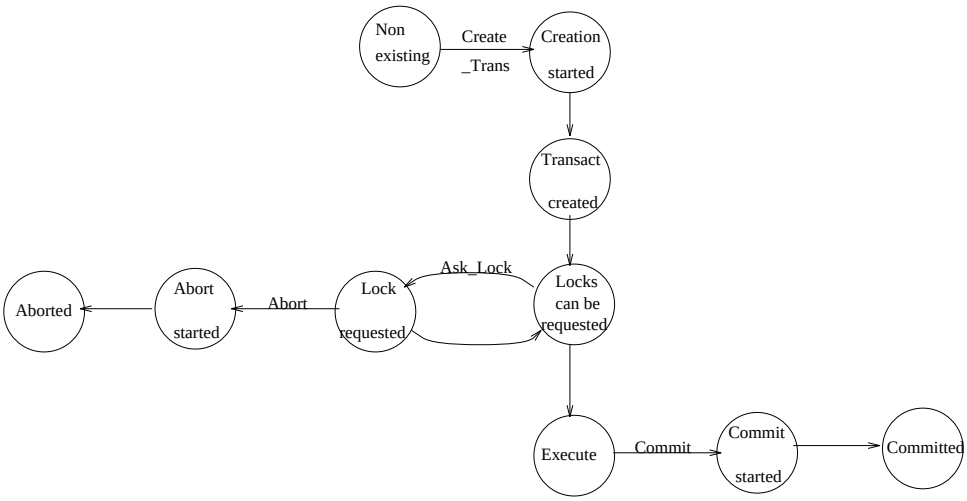


FIGURE 5.2.1. Kons Transaction: STD of the external behaviour.

In the state *Non existing*, the transaction is still not created. By using the transition Create_Trans, the operation is started and the state *Creation started* is reached. Eventually the

transaction is created and the state *Transact created* is reached.
The transaction can start to ask locks. If the lock request can't be accepted, because the lock request causes a conflict with an already obtained lock of another Kons Transaction, the transaction will be aborted. This is shown by the state *Abort started*. A Kons transaction always loses a conflict, when it requests a lock, held by another Kons transaction. If it is held by another type of transaction, the Kons transaction will win the conflict.
If the locks request is accepted, the lock is obtained and the transaction can request more locks.
Only in the beginning, the transaction asks for locks. When all the locks are obtained, it will start to execute. This means that some values in the database are changed with or without the use of tools. When the transaction is finished executing, the transaction will be committed.

The transaction of the type Auto is also initiated by the Process Engine. Here are the operations of the class Auto Transaction.

Create_Trans
Commit
Abort
Ask_Lock

Auto Transaction has the same export operations as Kons Transaction. The external behaviour of Auto Transaction is shown in figure 5.2.2. The external behaviours of Auto Transaction and Kons Transaction are pretty similar, as one can see.

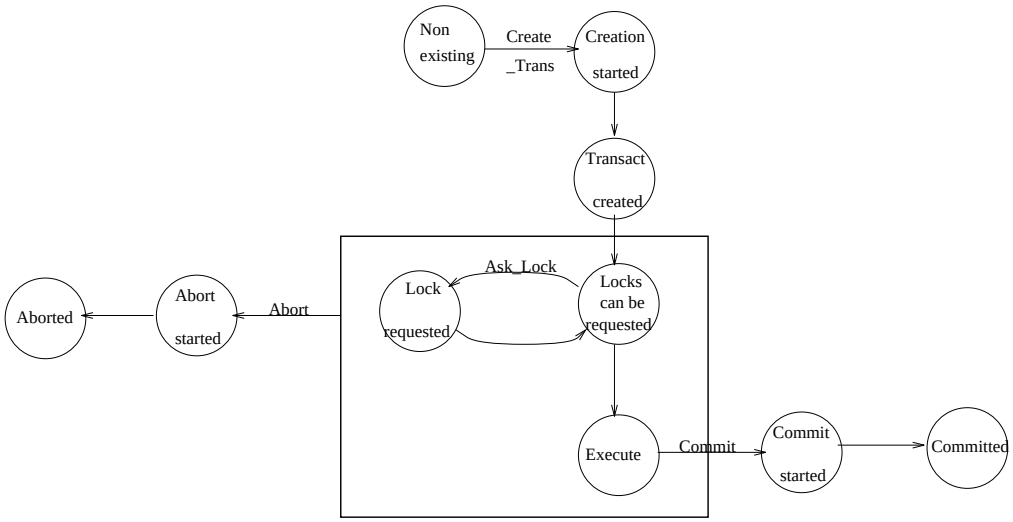


FIGURE 5.2.2. Auto Transaction: STD of the external behaviour.

The difference is that now not from one state but from three states the state *Abort started* can be reached.
Similar with the previous external behaviour of Kons Transaction, a transaction of the type Auto, called T1, can be aborted when it is requesting a lock, which is already set by another transaction, called T2 (state *Lock requested*). When the priority of the transaction T1 is lower then the priority of transaction T2, the lock won't be obtained and the Auto transaction T1 has to be aborted.
When a transaction T2 with a higher priority is requesting a lock, which has been set by the

transaction T1 of the type Auto, the lock has to be released by T1. This can happen in the states *Locks can be requested*, *Execute* and *Lock requested*. The transaction T1 then has to be aborted, because its lock is released premature.

Before the description of the other external behaviours is given, an important remark will be mentioned once more.
The instances of a Kons Transaction and Auto Transaction are always child transactions. The process engine starts such a child transaction, after the engineer has executed his parent transaction but before this parent transaction is committed. The process engine starts such child transactions to react on the changes in the data, which is caused by the work of the engineer. How this happens and how a child transaction influences its father will become more clear, when the external behaviour of the subclasses of Engineer initiated Transaction are described.

Next, the external behaviour of the class Pess_Akt Transaction is described. First, of course, the export operations are listed and in figure 5.2.3 the state transition diagram is shown.

Create_Trans
Commit
Abort
Ask_Lock

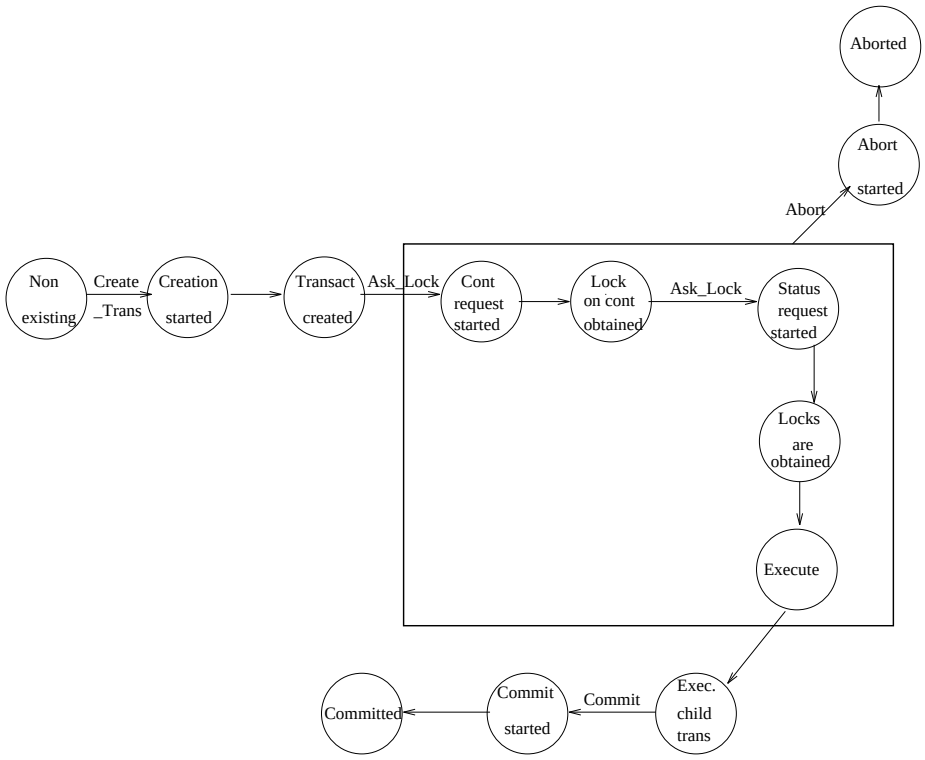


FIGURE 5.2.3. Pess_Akt Transaction: STD of the external behaviour.

After the transaction is created (state *Transact created*) the transaction always first asks for a lock on the contents and then on the status of the document. When all the locks are obtained, the state *Locks are obtained* is reached. The transaction can start to execute.
If the Pess_Akt Transaction T1 requests a lock and it can’t obtain this lock, then this transaction T1 has to be aborted (the transaction goes from either state *Cont request started* or state

Status request started to state *Abort started*).
It is possible that a lock request from another transaction of type T2 will get a lock, which is held by this Pess_Akt transaction T1. This happens when the priority of transaction T2 is higher then the priority of a Pess_Akt transaction. The lock will be released and transaction T1 can be aborted. This can only happen in the states *Cont request started, Lock on cont obtained, Status request started, Locks are obtained* and *Execute*.
At the end of the transaction, when the engineer has finished working on the document, child transactions can be started before the transaction is committed. The behaviour is then in the state *Execute child transaction*. Those child transactions are of the class Auto Transaction or Kons Transaction. Transactions of the type Auto and Kons can't initiate child transactions. Whether such a child transaction aborts or commits is not important for the execution of the Pess_Akt Transaction. After a child transaction is ended, a new child transaction can be started or the parent transaction can be committed. A child transaction has the possibility to inherit the locks from its parent.
If a lock, which is from a parent and its child, has to be released, because a transaction with a higher priority wants this lock, then both the child and the parent release the lock. The parent transaction will not be aborted, in this case.
By committing the (parent) transaction, the state *Commit started* is reached.

In figure 5.2.4, the STD of the external behaviour of the class Pess_AF Transaction is given. The export operations of the class Pess_AF Transaction are the following:

Create_Trans
Commit
Abort
Ask_Lock
End_Lock

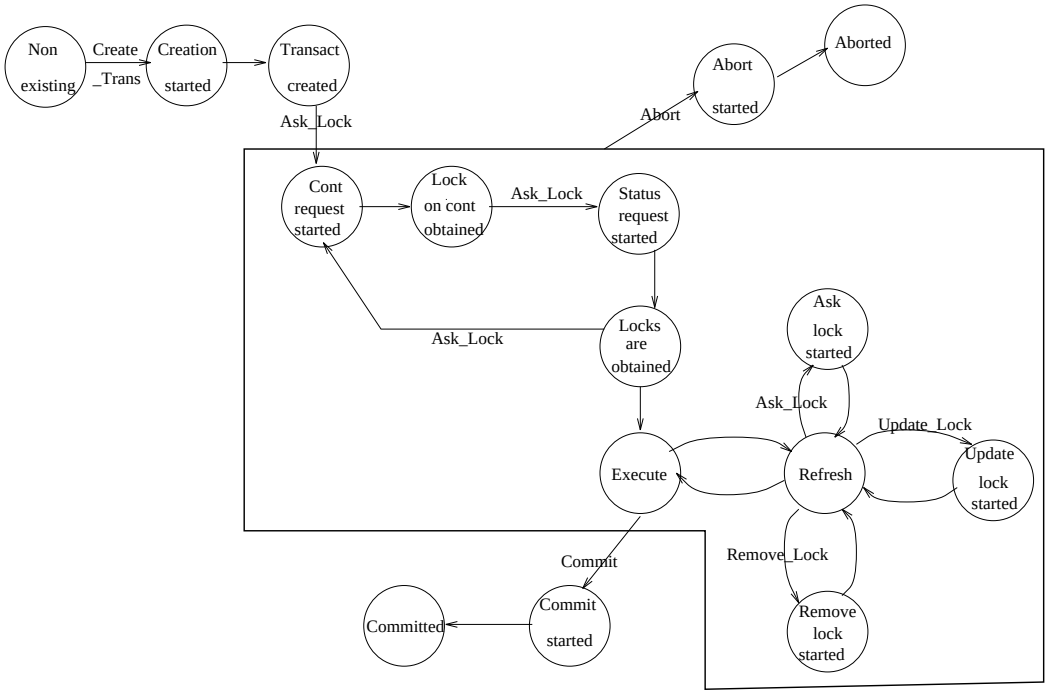


FIGURE 5.2.4. Pess_AF Transaction: STD of the external behaviour.

The external behaviours of Pess_Akt- and Pess_AF Transaction have some similarities. There are two main differences between the external behaviours.
First, a transaction of the type Pess_Akt locks the contents and status of one document. A transaction of the type Pess_AF can lock more then one document. A transaction of the type Pess_AF locks all the contents and status of the documents in the working context of the engineer.
Second, a transaction of the type Pess_AF anticipates when the working context of the engineer is refreshed. This means, that during execution, it can ask for new locks and release some already obtained locks, depending on the changes in the working context. All the contents and status, which are locked, will be saved in the data base. Because of this refreshing, child transactions will be started when a tool is stopped. This means, that the execution of child transactions is a part of the execution of the Pess_AF transaction. Therefore the state *Execute child transaction* is vanished in the above behaviour.
Furthermore also this transaction can be aborted by not getting a requested lock (transaction goes from state *Cont request started*, *Ask lock started* or *Status request started* to state *Abort started*). A transaction of the type Pess_AF can also be aborted, by releasing a conflicting lock, which is requested by another transaction with a higher priority (states *Execute*, *Remove lock started, Ask lock started, Update lock started, Refresh, Locks are obtained, Cont request started, Lock on cont obtained* and *Status request started*).
After execution, the transaction can start child transactions and will eventually commit.

The fifth and last subclass of Process Transaction is the class Opt_Akt Transaction. The export operations of this class are the following.

Create_Trans
Commit
Abort
Set_TS
Validate

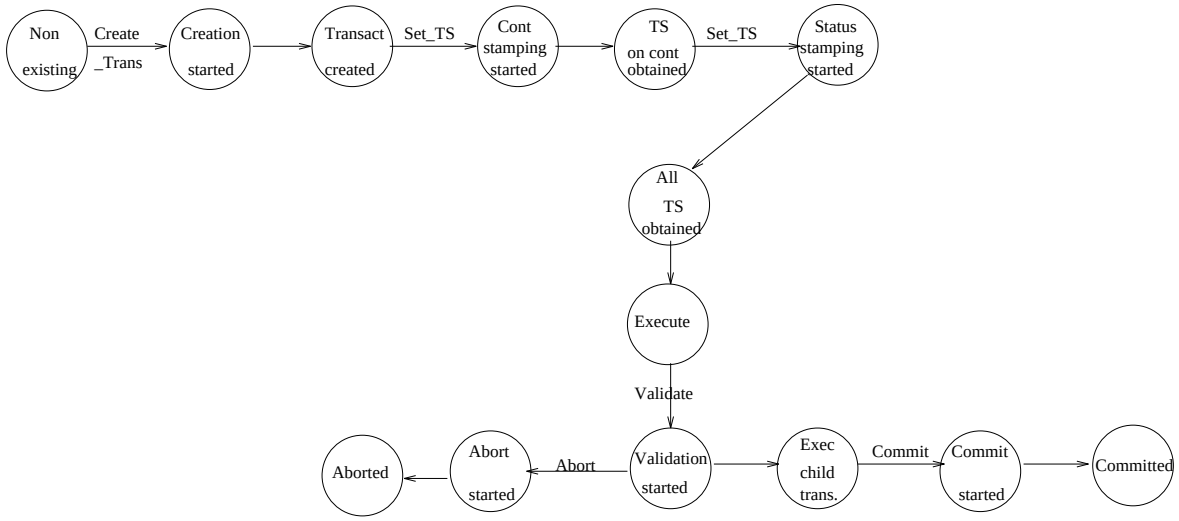


FIGURE 5.2.5. Opt_Akt Transaction: STD of the external behaviour.

The external behaviour of class Opt_Akt Transaction has some similarities with class Pess_Akt Transaction. The big difference is, that an Opt_Akt transaction sets time stamps and

a Pess_Akt transaction requests locks.
When the transaction is created (in state *Transact created*), the transaction sets time stamps. First one time stamp on the contents and then another time stamp on the status of a document is set.
After the time stamp have been set, the transaction is executing. Eventually, it will stop executing.
The transaction will then be validated. Validation means, it is checked whether a lock or a time stamp in the past or a lock, which is now obtained by a transaction, is in conflict with one of the time stamps of the Opt_Akt transaction.
If a time stamp of an Opt_Akt transaction is in conflict with another time stamp or lock, the transaction will be aborted (state Abort started is reached). The data used by this transaction could be inconsistent.
If the transaction is validated positive, all the time stamps are cleaned and changed in locks. The type of the transaction is changed from Opt_Akt to Pess_Akt. Then, the transaction can start child transactions and will eventually be committed

This finishes the discussion of the external behaviours of the five Process Transaction classes. Now the external behaviour of the class Time Stamp is described.

First the list with export operations is given. In figure 5.2.6, the external behaviour of the class TimeStamp is shown.

Create_TS
Clean_TS

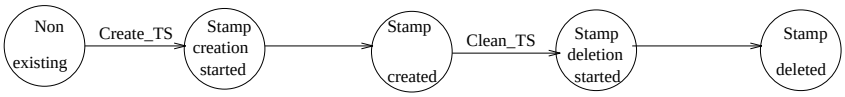


FIGURE 5.2.6. TimeStamp: STD of the external behaviour.

When an Opt_Akt transaction is started, it will need time stamps. By using the operation Create_TS, creation of a time stamp is started. This means that the external behaviour changes from state *Non existing* to state *Stamp creation started.*
Eventually, the time stamp is not needed any more. The Clean_TS operation is started to delete the time stamp (state *Stamp deletion started* is reached).

The next external behaviour is from class Log. The list of the export operations of the class Log is:

Add_Tuple_in_Log
Conflict_with_Log?

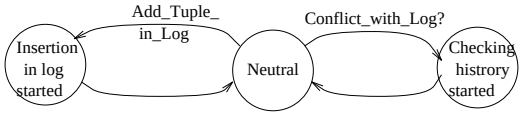


FIGURE 5.2.7. Log: STD of the external behaviour.

If a transaction of the type Opt_Akt is validating, it checks in the log whether the time stamp, which is now valuating, is not in conflict with an already released lock or time stamp. Furthermore, the log inserts tuples in its contents.

Next, the external behaviour of class Tuple is given. The list of the export operations of the class Tuple is:

Create_tuple
Insert_tuple
Conflict_with_Tuple?
Delete_tuple

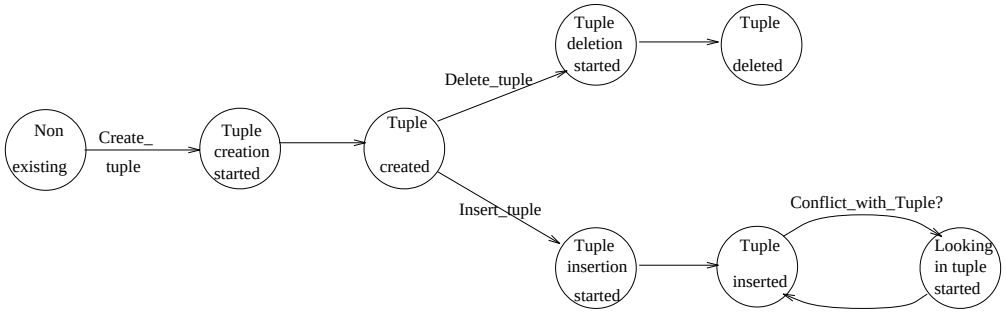


FIGURE 5.2.8. Tuple: STD of the external behaviour.

If a transaction is committing or refreshing, it puts information about its released locks in the log.
When a tuple is created, information from the lock is stored into the tuple. After it has been created, it is inserted in the log.

If a transaction is validating, it can put information about its cleaned time stamp(s) in the log. First, the time stamps are validated, until all the stamps are validated successfully or there is a conflict between this validating stamp and a time stamp or lock of another transaction.
When a time stamp is validated successfully, a tuple is created. Information from the time stamp is stored into the tuple, which is created.
If none of the time stamps were in conflict, all the tuples, created by this transaction, are inserted in the log. This means that the external behaviour(s) of the tuple(s) will go from state *Tuple created* to state *Tuple insertion started*.
If a time stamp is in conflict, all the tuples, which are created until now by this transaction, are deleted. This only happens when the transaction is validated unsuccessfully.

A validating Opt_Akt transaction can look in inserted tuples. An Opt_Akt transaction then checks whether its time stamps are in conflict with other already released locks or time stamps.

Now the external behaviour of the class Locks will be described. The list of export operations is:

Get_Write_Lock
Get_Read_Lock
Get_Lock_Father
Release_Lock
Compatibility
Conflict

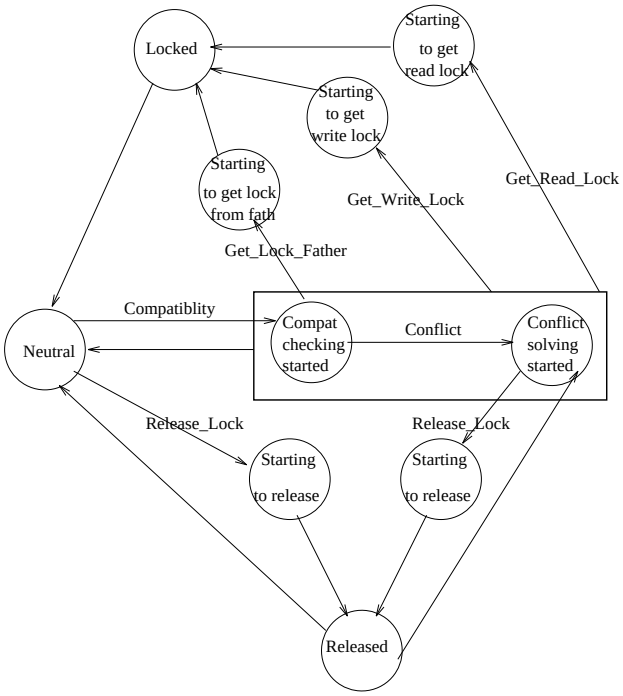


FIGURE 5.2.9. Locks: STD of the external behaviour.

When nobody wants to release or ask for a lock, the behaviour is in the state *Neutral*. In this state, zero or more locks can be set on this data (contents or status of a document).
If a transaction only wants to release its lock, it asks for the operation Release_Lock and reaches the state *Starting to release*.
If a transaction requests a lock, it first checks whether the transaction(s), which hold this lock and the requesting lock are compatible. If for instance no transactions hold a lock on this contents or status, then the request is compatible. If a child requests for a lock, which is already hold by its father, then the request is also compatible. The father can have for instance a write-lock and the child can then also request and obtain this write-lock, although two write-locks on the same contents or status are usually incompatible.
If the requesting lock and the transaction(s), which hold the lock aren't compatible, then there is a conflict. Either the requesting lock won't be accepted (external behaviour then goes to state *Neutral*) or the transaction(s), which hold the lock, have to release this lock (external behaviour goes to state *Starting to Release*). This depends on the priority of the involving transactions.
When the request is either compatible or wins the conflict, a lock on a contents or status will be set.

The Opt_Akt transaction also uses the class Lock. When the Opt_Akt transaction is validating, it has to know whether locks, which are already set, are incompatible with its time stamps. If the time stamps are incompatible, then the Opt_Akt transaction also won't win the conflict. If the time stamps are compatible, then the time stamps are changed into locks.

The next class, of which the external behaviour is described, is the class Process Engine. The export operations are:

Start_WC
Stop_WC
Start_Act
Stop_Act
Refresh

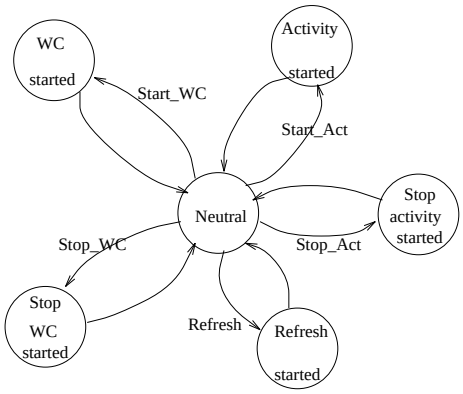


FIGURE 5.2.10. Process Engine: STD of the external behaviour.

The process engine controls the creation of the working context. A working context can be created with pessimistic protection. This means that a Pess_AF transaction is initiated and that the contents and status of all documents in the working context will be locked. Such a pessimistic protection is chosen by either the process engine or the user.
If all the contents and status of the working context have to be locked, but the Pess_AF transaction loses a conflict with another transaction, which already holds a lock on a contents or status of this working context, the working context won't be created.
The engineer can start interactive activities. This means that an interactive tool is started. When the working context isn't build with a Pess_AF transaction, an Opt_Akt or Pess_Akt transaction will be created and the tool is started within this transaction.
On one moment, an engineer can work with several interactive tools. Before the working context is stopped, all the interactive activities of the engineer have to be stopped.
The working context can also be refreshed by the process engine. Refresh means, that some documents of the working context will leave the working context and some new ones will appear in the working context. If the working context is build with pessimistic protection, the Pess_AF transaction also has to refresh its locks.

The export operations of the class Working Context are:

Create_WC
Delete_WC
Refresh_Display

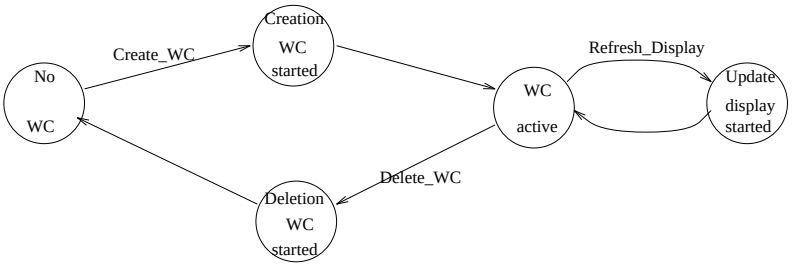


FIGURE 5.2.11. Working Context: STD of the external behaviour.

The working context will first be created. When the working context is created, the display can be updated. The display is only updated, when the process engine refreshes the working context. The new display will show the new contents and status of the working context.
When the working context isn't needed any more, it is deleted.

The export operations of the class Contents are:

Start_Inter_Tool
Stop_Inter_Tool
Use_Batch_Tool
Save_Copy
Create_Copy
Copy_in_Privat

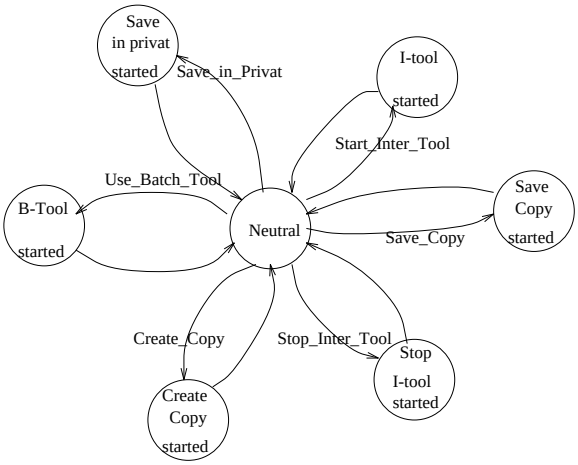


FIGURE 5.2.12. Contents: STD of the external behaviour.

An interactive - or a batch tool can be used, to work on a document. The transactions of the type Pess_Akt, Pess_AF and Opt_Akt only support interactive tools. Transactions of the type Auto and Kons only support batch tools.
An Auto or Kons transaction only support one batch tool at a certain point in time. One opera-

tion is used to control the batch tool.
The beginning and ending of an activity (and therefore a tool) are directly initiated by the engineer. For this reason, two operations are used to control the interactive tool, one operation to start the tool and one operation to stop the tool.
Copies of the contents of a document can be created and saved. A copy is saved in a private directory, when the transaction, which supported the execution on this contents, is aborted. If the transaction will be committed, the copy will be saved in the data base.

The export operation for the class Status is:

New_Status

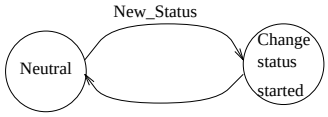


FIGURE 5.2.13. Status: STD of the external behaviour.

The operation New_Status makes sure, that the status of a document is changed in a new status.

This finishes the discussion of the external behaviours of all the classes. Next, it is indicated for each class, which (export) operations it imports from elsewhere.
After that, the internal behaviour of each operation of each class is specified.

5.3 Import list of the uses relationships

To know what export operations an internal behaviour may import from an other class, for each uses relationship of the import / export diagram in the data perspective (figure 5.1.4.) a list of probably imported operations is presented.

Uses3
Start_WC (User_Name, Role)

Uses5
Stop_WC (User_Name, Role)
Start_Act (User_Name, Role, Doc_Name, Activity)
Stop_Act (User_Name, Role, Doc_Name, Activity)
Refresh (User_Name, Role)

Uses6
Create_WC (User_Name, Role)
Delete_WC ()
Refresh_Display ()

Uses7
Create_Trans (Ident, Type, User_Name, Role, Ident_Father_Trans)
Commit ()

Uses8
Set_TS (Access_Type, Doc_Name, Object_Type)
Validate ()

Uses9
Ask_Lock (Access_Type, Doc_Name, Object_Type)

Uses10
Remove_Lock (Access_Type, Doc_Name, Object_Type)
Update_Lock (Access_Type, Doc_Name, Object_Type)

Uses11
New_Status (Doc_Status)

Uses12
Start_Inter_Tool (Ident)
Stop_Inter_Tool (Ident)
Use_Batch_Tool (Ident)

Uses13
Delete_WC ()

Uses14
Create_Copy (Ident)
Save_Copy (Ident)
Copy_in_Privat (Ident)
Stop_Inter_Tool (Ident)

Uses15
Abort ( )

Uses16
Create_TS (TS_List, Access_Type, Object_Type, Doc_Name, Time)
Clean_TS (TS_List, Ident)

Uses17
Get_Write_Lock (Object_Type, Ident, Lock_List)
Get_Read_Lock (Object_Type, Ident, Lock_List)
Get_Lock_Father (Access_Type, Object_Type, Ident, Ident_Father_Trans, Lock_List)
Release_Lock (Object_Type, Acces_Type, Ident, Lock_List)
Conflict (Object_Type, Type)
Compatibility (Object_Type, Access_Type)

Uses18
Create_tuple (Object_Type, Access_Type, Doc_Name)
Insert_tuple (End_of_Log_Contents)
Delete_tuple ()

Uses19
Conflict_With_Log? (Doc_Name, Access_Type, Object_Type, Time)
Add_Tuple_in_Log (Time)

Uses20
Conflict_With_Tuple? (Doc_Name, Access_Type, Object_Type, Time)

With use of the list of the imported operations, the internal behaviours of each export operation can be specified. Specifying the internal behaviour for every export operation of each class is the next step.

5.4 Internal behaviour

For each export operation, its internal behaviour is specified in this section. The name of an internal behaviour is composed of the name of the corresponding export operation and the prefix ‘int-’.
Within the internal behaviour of an export operation two different types of operations can occur.
The first type of operation in an internal behaviour is the from elsewhere imported operation. When this occurs, the convention is to add ‘call_’ as a prefix to the name of the imported operation from somewhere else. It is a reminder that the transition labelled with the corresponding exported operation as well as the internal behaviour of the same exported operation are not necessarily synchronized, i.e. taking place at the same time of the call. This means that after the call is made, the internal behaviour can start to execute the next operation, without waiting for the calling operation to finish.
The second type of operations consists of those without the prefix ‘call_’. These are merely internal operations. Those having the prefix ‘act_’ reflect some internal communication between an internal behaviour and the external behaviour. It denotes the preliminary activation of the internal behaviour as a whole, while it is not actually going. This can be compared to a computer program already submitted to the operating system of a computer without having started its execution: it has only been scheduled.

First, the internal behaviours of the export operations of class Tuple are specified. Before the internal behaviours of the operations of class Tuple are given, the names of the export operations are listed in the same order as the internal behaviours are specified.

Create_tuple
Insert_tuple
Delete_tuple
Conflict_with_tuple?

The first internal behaviour of an export operation of class Tuple is:
Create_Tuple (Object_Type, Acces_Type, Doc_Name)

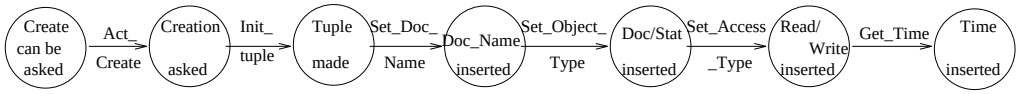


FIGURE 5.4.1. int-Create_Tuple: STD of the internal behaviour.

A tuple is created, when a transaction releases a lock, because it is committing or refreshing, when a lock is updated during refreshing and when an Opt_Akt transaction cleans a time stamp.
All the relevant information about this lock or time stamp is inserted in the tuple. In the tuple is also the time of its creation stored.
As already mentioned in the data perspective, the functionality of a tuple can be compared with the INSERT operation of Stefan Wolf’s dissertation. As one can see on page 152 of the dissertation, not four but five information items are inserted in the log. The identifier of the transaction is also inserted. This is however irrelevant to put this information in the log and

therefore this is left out in this specification.

The second operation of the class Tuple is:

Insert_Tuple (End_of_Log_Contents)

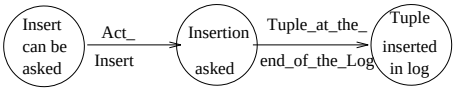


FIGURE 5.4.2. int-Insert_Tuple: STD of the internal behaviour.

Once the tuple is created, it can be inserted in the log or deleted. A tuple is inserted in the log, when a transaction is committing or refreshing or when an Opt_Akt transaction is validated successfully.
If it is inserted in the log, the tuple is added at the end of the log.

The third operation of the class Tuple is:

Delete_Tuple ( )

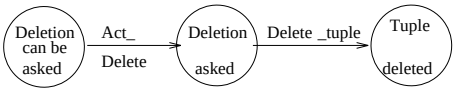


FIGURE 5.4.3. int-Delete_Tuple: STD of the internal behaviour.

When an Opt_Akt transaction is validating and one of its time stamps is in conflict, all the already created tuples of its successfully validated time stamps have to be deleted. The tuples, which are deleted, were not yet inserted in the log.

The fourth and last operation of the class Tuple is:

Conflict_with_Tuple? (Doc_Name, Access_Type, Object_Type)

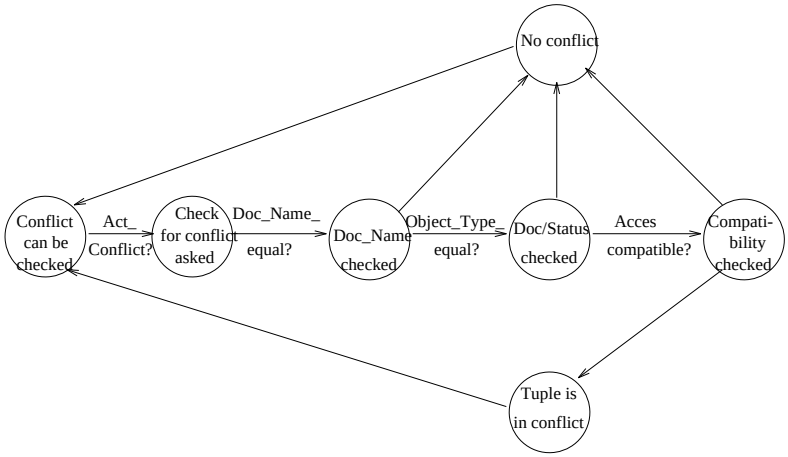


FIGURE 5.4.4. int-Conflict_with_Tuple?: STD of the internal behaviour.

An Opt_Akt transaction checks whether its time stamp is in conflict with a tuple.
First, it is checked whether the tuple and the time stamp accessed the same document. The next check is whether both the tuple and the time stamp had a lock on the status or contents of the same document. The last check is to compare the access. It is checked whether the access of the tuple and the time stamp are incompatible.
If all the checks are positive, the time stamp is in conflict with the tuple.

This concludes the specification of the internal behaviours of class Tuple. Now, the internal behaviour of the operations of class Log is given. The operations of this class are:

Conflict_with_Log?
Add_Tuple_in_Log

The first name of the export operation of class Log is :
Conflict_with_Log?(Doc_Name, Access_Type, Object_Type, Time)

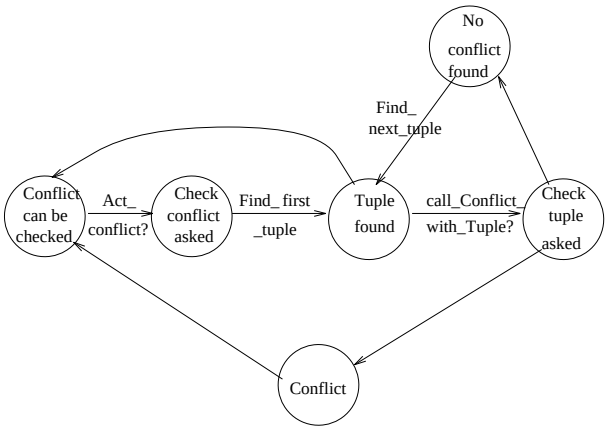


FIGURE 5.4.5. int-Conflict_with_Log?: STD of the internal behaviour.

An Opt_Akt transaction checks whether one of its time stamps is in conflict with a tuple in the log.
When the internal behaviour is activated, it first searches for the first tuple in the log, which was created just after this Opt_Akt transaction created the time stamp, it is now validating. If there is no tuple, the time stamp is not in conflict. Otherwise, it is checked whether this tuple is in conflict with the time stamp.
If there is no conflict between the time stamp and the tuple, the next tuple is retrieved from the log. When no new tuples can be retrieved any more, it is concluded that the time stamp isn't in conflict with a tuple.
If there is a conflict between a tuple and a time stamp, the time stamp is validated unsuccessfully.

The second export operation of class Log is:

Add_Tuple_in_Log (Time)

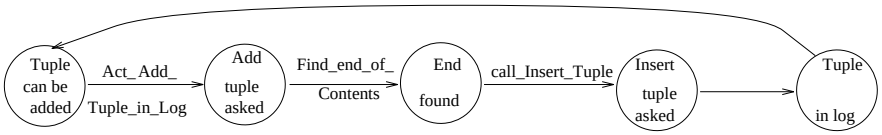


FIGURE 5.4.6. int-Add_Tuple_in_Log: STD of the internal behaviour.

An already created tuple will be added to the log. First, the end of the log is found. At the end of the log, the new tuple is inserted.

Next, the internal behaviours of the operations of the class TimeStamp are specified. First the operations are listed.

Create_TS
Clean_TS

The first operation of the class TimeStamp is:

Create_TS (TS_List, Access_Type, Object_Type, Doc_Name, Time)

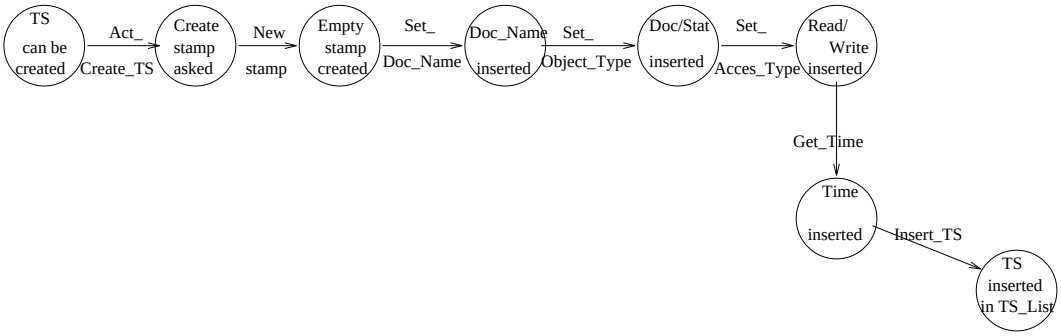


FIGURE 5.4.7. int-Create_TS: STD of the internal behaviour.

When a time stamp will be created, its attributes receive a value. So, in the internal behaviour, the attributes Doc_Name, Object_Type and Access_Type are set. In the attribute Time, the time of creation of the time stamp is inserted. Moreover, this time stamp is added to the list of time stamps, TS_List, of the Opt_Akt transaction.

The second and last operation of class TimeStamp is:

Clean_TS (TS_List, Ident)

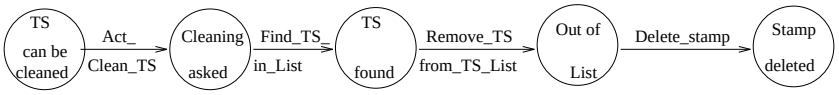


FIGURE 5.4.8. int-Clean_TS: STD of the internal behaviour.

The time stamp has to be cleaned (deleted). This time stamp is removed from the list of the Opt_Akt Transaction and deleted.

Now the internal behaviour of the export operations of the class Lock will be described. The operations of the class Lock are:

Compatibility
Conflict
Get_Write
Get_Read
Get_Father
Release_Lock

The first operation of class Lock is:

Compatibility (Object_Type, Access_Type)

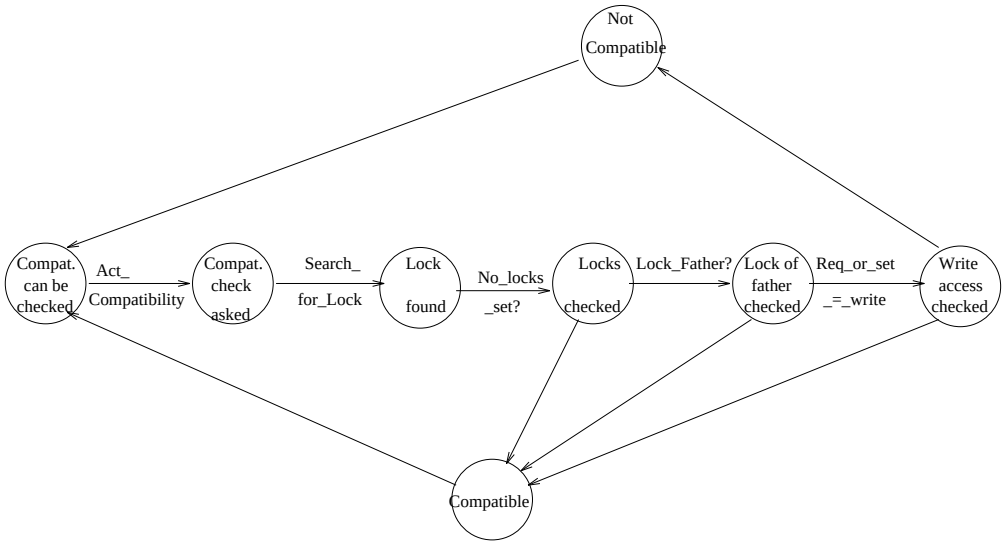


FIGURE 5.4.9. int-Compatibility: STD of the internal behaviour.

A transaction uses this operation to check, whether its lock request is compatible with possibly obtained lock(s) on the contents or status of a document. This specific contents or status is called data X.

First, it is checked, whether there is no lock on data X. If there is no lock on data X, the request is compatible.
Otherwise, it is checked whether the father (if this transaction has a father) of the requesting transaction already has a lock on data X. If this is the case, the child transaction now inherits the lock of its father. A child only inherits a lock from its father, when it needs a lock from its father.
Otherwise, it is checked whether either the transaction(s), which already obtained the lock, have set a write lock or the requesting transaction wants a write lock. If this is the case, the requested lock isn't compatible with the already obtained lock(s) on data X and there is a conflict.

If none of the involving transactions already have a write lock or request a write lock, the request is compatible.

The second operation of class Lock is:

Conflict (Object_Type, Type)

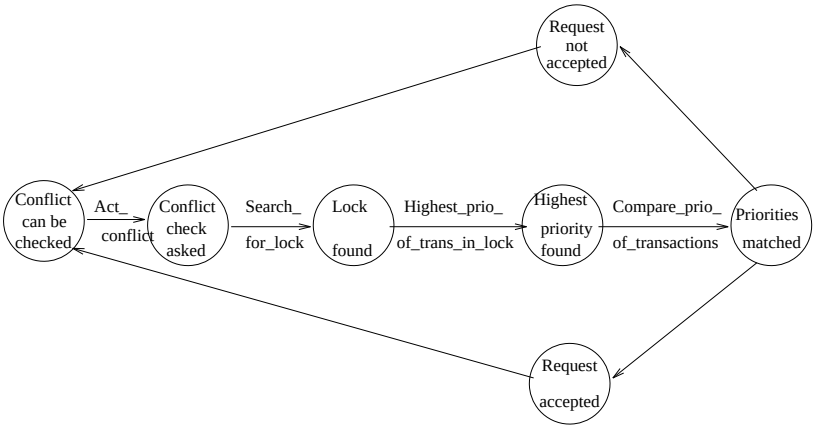


FIGURE 5.4.10. int-Conflict: STD of the internal behaviour.

When it is found out that a lock request is not compatible, this conflict has to be solved.

First the transaction with the highest priority, which holds the conflicting lock, is traced and found.
After that, the priority of this transaction with the highest priority and the priority of the requesting transaction are compared.
If the requested transaction has a higher priority, it wins the conflict and the lock request is accepted.
Otherwise, it loses the conflict and the lock request is rejected.

Priority depends on the type of the transaction. These priority rules are already given in the chapter Process Transactions.

The third operation of class Lock is:

Get_Write(Object_Type, Ident, Lock_List)

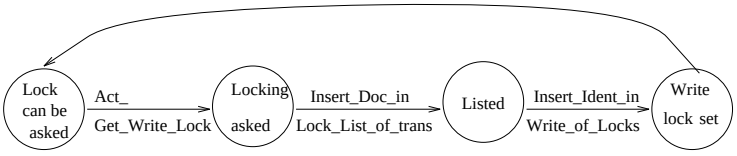


FIGURE 5.4.11. int-Get_Write_Lock: STD of the internal behaviour.

A transaction gets a write lock. Therefore its attribute Lock_List has to be updated. Then the lock will be really set, by inserting the value Identifier in the attribute Write of class Lock.

The fourth operation of class Lock is:

Get_Read(Object_Type, Ident, Lock_List,)

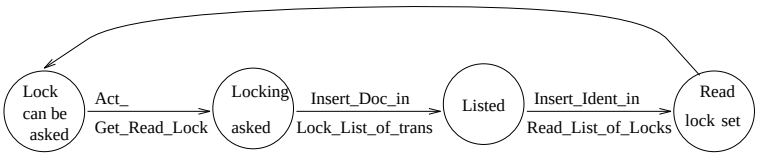


FIGURE 5.4.12. int-Get_Read_Lock: STD of the internal behaviour.

In the list of locks of the transaction is this lock added. Then a read lock is set, by inserting the attribute Identifier.

The fifth operation of class Lock is:

Get_Father(Access_Type, Object_Type, Ident, Ident_Father_Trans, Lock_List)

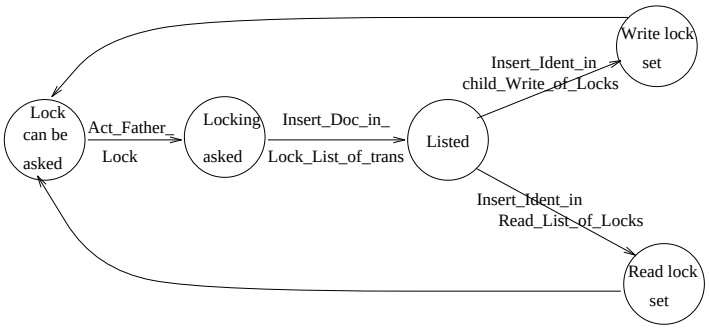


FIGURE 5.4.13. int-Get_Lock_Father: STD of the internal behaviour.

A child transaction needs a lock, which is already held by its father. The child inherits this lock.
First the list of locks of the child transaction is updated.
Then a write or a read lock is set. If the child wants a write lock, this is specially marked. This is done, because normally two write locks are incompatible. If a parent transaction already holds a write lock, it gives its child the opportunity to set the write lock and therefore to add its value of the Identifier to the attribute Write.

The sixth and last operation of the class Lock is:

Release_Lock (Object_Type, Access_Type, Ident, Lock_List)

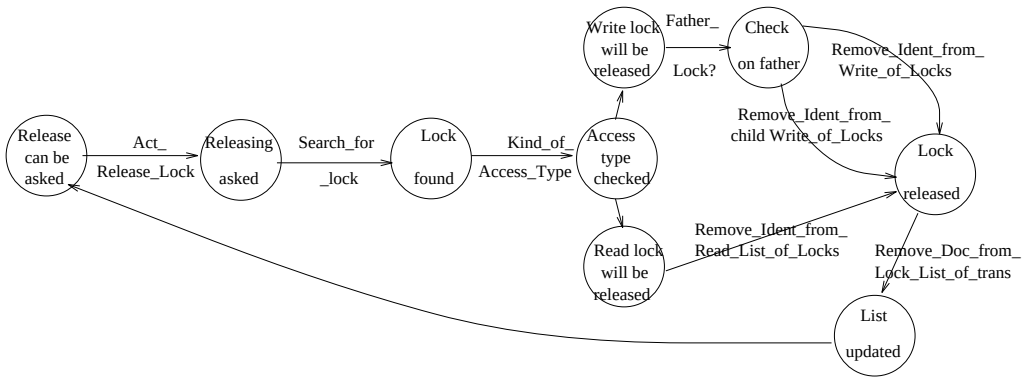


FIGURE 5.4.14. int-Release_Lock: STD of the internal behaviour.

To release a lock, the right contents or status has to be found.
Then a write or read lock has to be released. If a read lock will be released, the value Identifier is removed from the attribute Read_List. If a write lock will be released, it is checked whether this lock is inherited by a child transaction or not. In both cases, the attribute Write is updated. The attribute Write can store two Identifiers. If a child releases its write lock, the child value will be cleaned. Otherwise the normal value will be cleaned.
At the end, the attribute Lock_List of class Process Transaction, which contains information over all the obtained locks of a transaction, is updated.

The internal behaviour of operations of the superclass Process Transaction will now be specified. The operations of its subclasses are examined afterwards. First, the operations of the class Process Transaction are listed.

Create_Trans
Commit
Abort

The first operations of the superclass Process Transaction is:

Create_Trans (Ident, Type, User_Name, Role, Ident_Father_Trans)

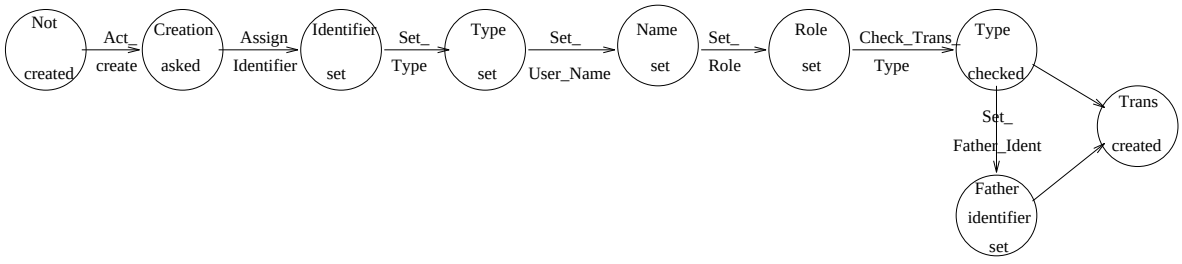


FIGURE 5.4.15. int-Create_Trans: STD of the internal behaviour.

When a transaction is created, its attributes have to be initiated. The attribute value Identifier is assigned to this transaction. The values for Type, User_Name and Role are given by the process engine. The process engine also gives a value for the attribute Ident_Father_Trans, if this

transaction is of the type Auto or Kons.
A specific transaction can of course be created just one time. So, no incoming transitions come in the state *Not created*. After the transaction is created, this state will never be entered any more.

The second operation of the class Process Transaction is:

Commit ( )

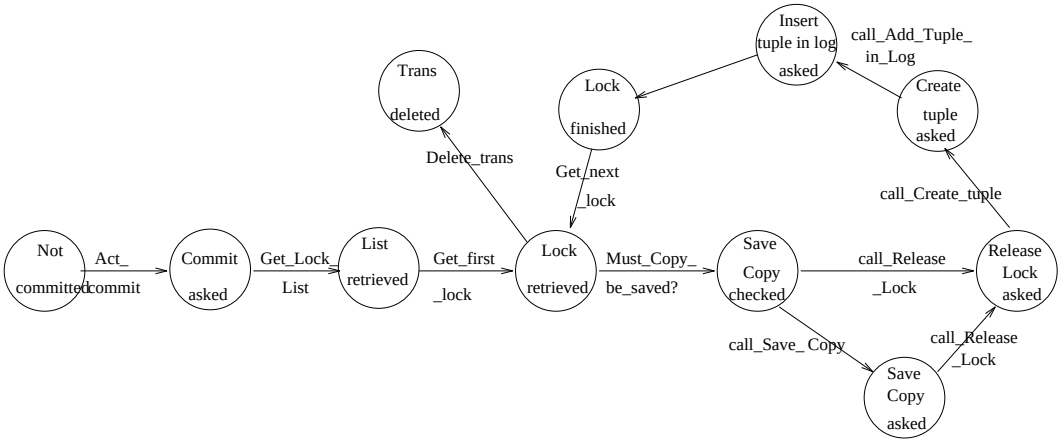


FIGURE 5.4.16. int-Commit: STD of the internal behaviour.

First, the attribute Lock_List, with the information concerning all the locks held by this transaction, is retrieved.
Then a lock from the list is taken and if the information in the copy of a contents is changed, it is saved. Now the lock can be released, by calling the operation Release_Lock from the class Locks. Then the information concerning this lock is stored in the log. Therefore, first a tuple will be created and with the relevant lock information, it is inserted in the log. Then, if other locks still exist for this transaction, they will also be taken out of the list and be released. If all the locks are released, the transaction is deleted.
It can happen, that no locks are any more in the list, when the transaction starts committing. This happens for instance, when all the locks had to be released, when the child transactions were executing. Then, the behaviour will immediately go from state *Lock retrieved* to state *Trans deleted*.

The third operation of the class Process Transaction is:

Abort ( )

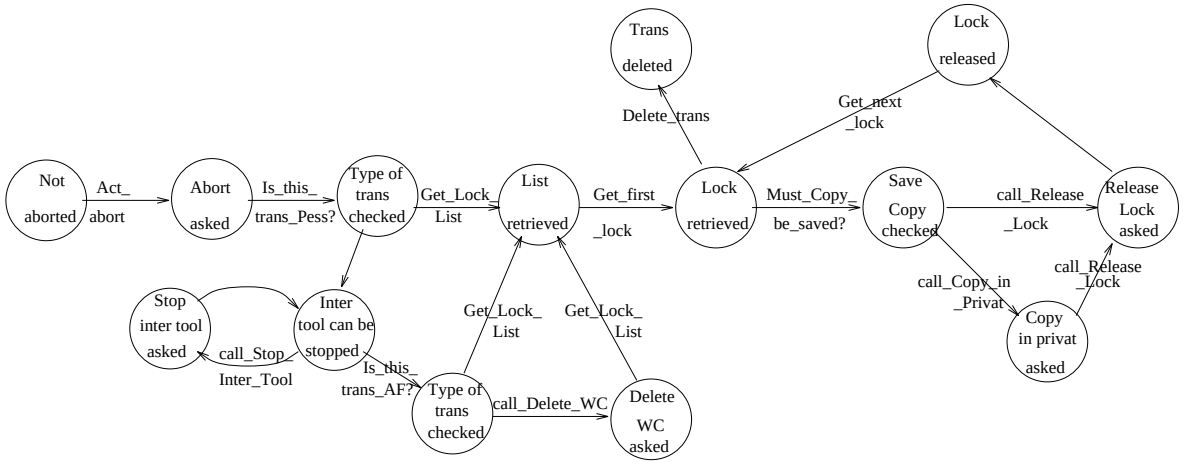


FIGURE 5.4.17. int-Abort: STD of the internal behaviour.

First, it is checked what type of transaction will be aborted. If it is a transaction of the type Pess_Akt or Pess_AF, active tools and the working context may be affected. Interactive tools, which execute with protection of this transaction have to be stopped. After that, if the transaction, which will be aborted, is a Pess_AF transaction, the working context also has to be deleted.
If the transaction is of the type Opt_Akt, Auto or Kons and has to be aborted, the stopping of tools is either initiated somewhere else or must not be specifically initiated.

Now, the transaction, which will be aborted, has to release all its locks. It retrieves the list concerning all the locks held by this transaction and then releases all locks from this list. If a lock is on a contents and it is changed, the contents is copied into a private directory.
It can happen, that no locks are hold by an aborting transaction. This is the case, when an Opt_Akt transaction is unsuccessfully validated and that no time stamps had yet been changed in locks.

This concludes the specification of the internal behaviours of operations of class Process Transaction. Next, the operation of the class Pessimistic Transaction is specified. The operation of this class is:

Ask_Lock (Access_Type, Doc_Name, Object_Type)

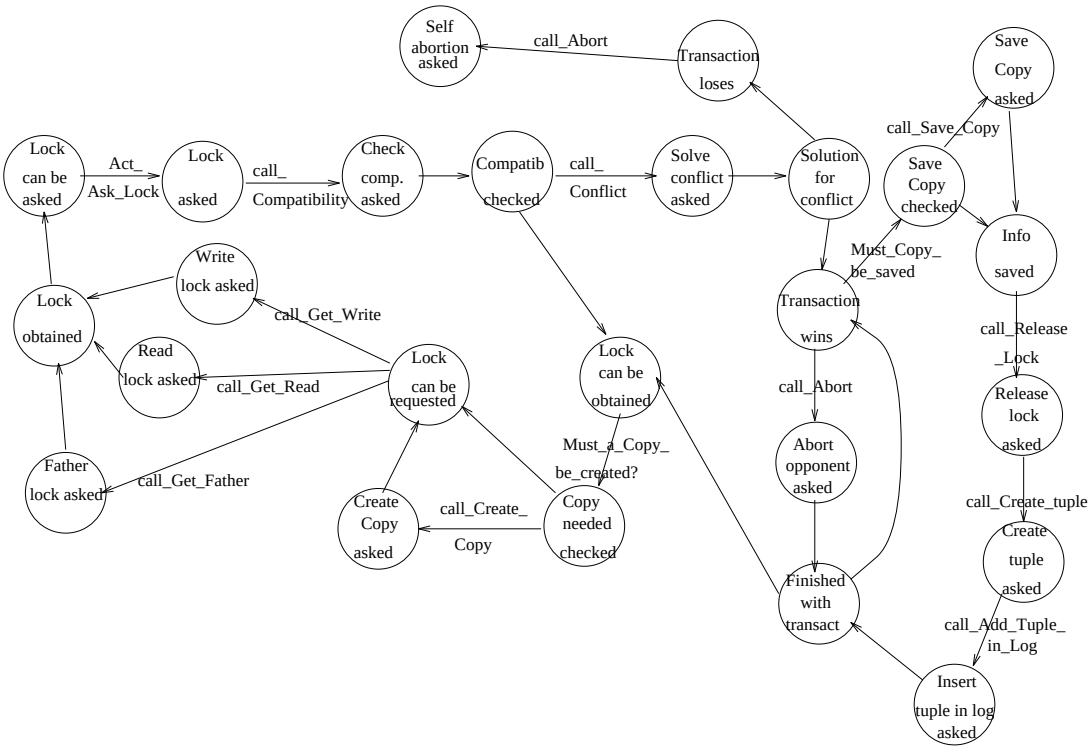


FIGURE 5.4.18. int-Ask_Lock: STD of the internal behaviour.

First the operation looks whether the lock request on a contents or status is compatible with the possibly obtained lock(s). If the request is compatible, the lock is obtained.

Otherwise, the conflict has to be solved.
If the request is rejected, the requesting transaction is aborted.
If the request is accepted, the transactions, which hold a lock on this contents or status, either are aborted or release this lock. When this lock will be released, the information is saved, if changed and a tuple is created and together with the information of this lock, inserted in the log.
After the conflicting transaction(s) have released the lock (and are perhaps aborted), the lock can be obtained by the requesting transaction. Before the lock is really obtained, it is first checked whether a copy has to be created. A copy is only created, when the contents of a document will be locked.
Either a write or a read lock is obtained. Furthermore, a child can inherit the lock of its father.

Now, the internal behaviours of the operations of the class Opt_Akt Transaction are specified. The operations of this class are:

Set_TS
Validate

The first operation of the subclass Opt_Akt Transaction is:.

Set_TS (Access_Type, Doc_Name, Object_Type)

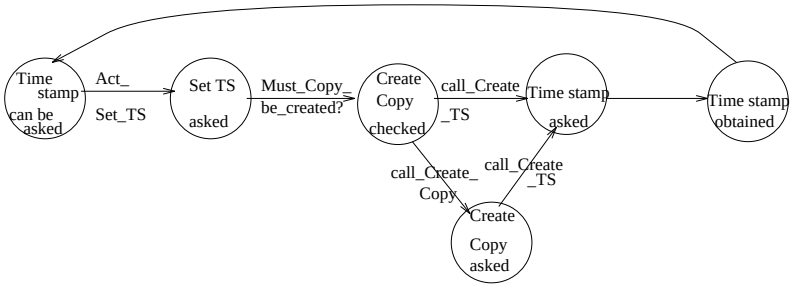


FIGURE 5.4.19. int-Set_TS: STD of the internal behaviour.

When an optimistic transaction asks for a time stamp, the request is always accepted. First, it is checked whether a copy has to be created. A copy will only be created if a contents of a document will be stamped. By calling the operation Create_TS, a read or write time stamp is set.

The second and last operation of subclass Opt_Akt Transaction is:

Validate ( )

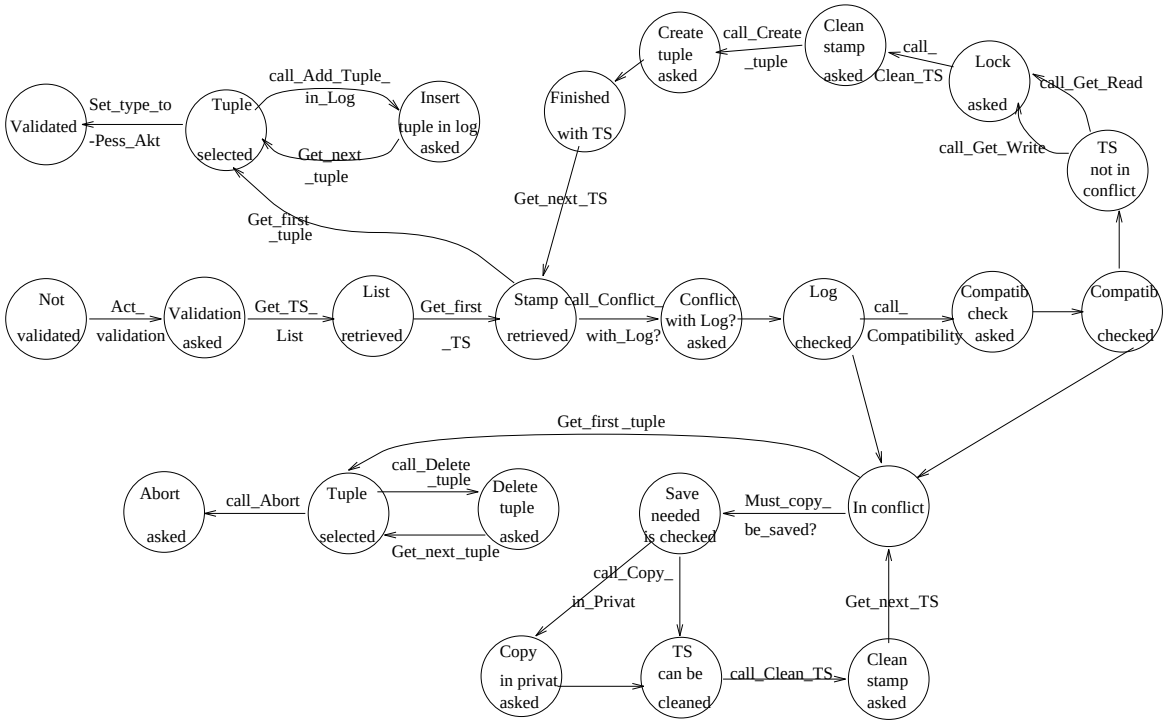


FIGURE 5.4.20. int-Validate: STD of the internal behaviour.

To validate all the time stamps, the list with the time stamps first has to be retrieved. Then the first time stamp of the list will be taken. For this time stamp, but also for the other time stamps from the list, it is first checked whether a conflicting time stamp or lock exists in the log. If not it is checked whether a transaction holds a conflicting lock.
If there is or was no conflicting lock, the stamp is changed in a lock. After that, the stamp is cleaned and a tuple with the information of the time stamp is created.
If more time stamps remain to be validated, the next stamp will be considered.
After successful validation of all time stamps, all the tuples are inserted in the log. At the end, the type of the transaction changes from Opt_Akt to Pess_Akt. Then, the transaction is successfully validated in to a Pess_Akt transaction, with locks.
If there exists a conflicting lock or time stamp, the transaction has to be aborted. First all time stamps, which are still not cleaned, will be deleted and if its contents is changed, it is saved in a private directory. Also the tuples are deleted, which have been created during the validation. So no information will be written in the log. The operation abort makes sure that all relevant information, already with a lock, will be copied into a private directory and releases the already obtained locks.

The operations of class Pess_AF Transaction are:

Remove_Lock
Update_Lock

The first operation of the subclass Pess_AF Transaction is:

Remove_Lock (Access_Type, Doc_Name, Object_Type)

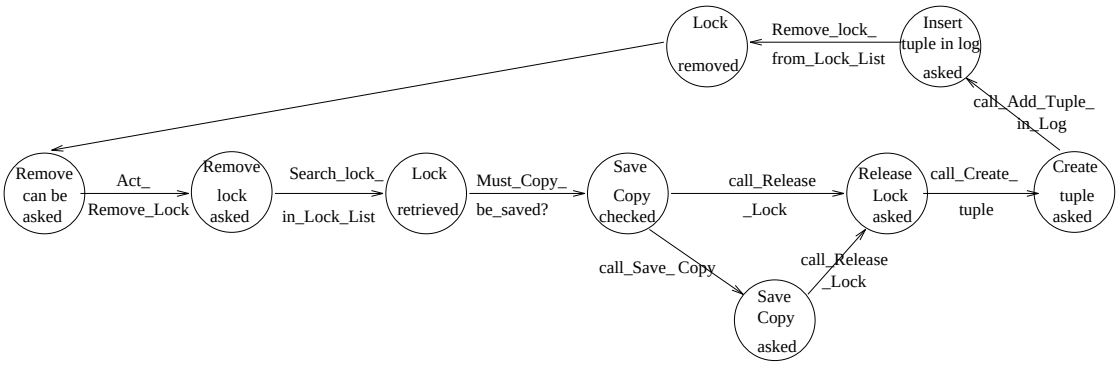


FIGURE 5.4.21. int-Remove_Lock: STD of the internal behaviour.

When a Pess_AF transaction is refreshing its working context, locks have to be released and requested. The above operation removes a lock from the Pess_AF transaction.
First, the operations searches for the right lock. It is checked whether the data, on which this lock is set, has to be saved. Only a contents, which has changed, has to be saved in the data base. Now the lock can be released and the Log can be updated. At the end, the attribute Lock_List of the transaction is updated, because this lock is no longer obtained by the transaction.

The second and last internal behaviour of class Pess_AF Transaction is:

Update_Lock (Access_Type, Doc_Name, Object_Type)

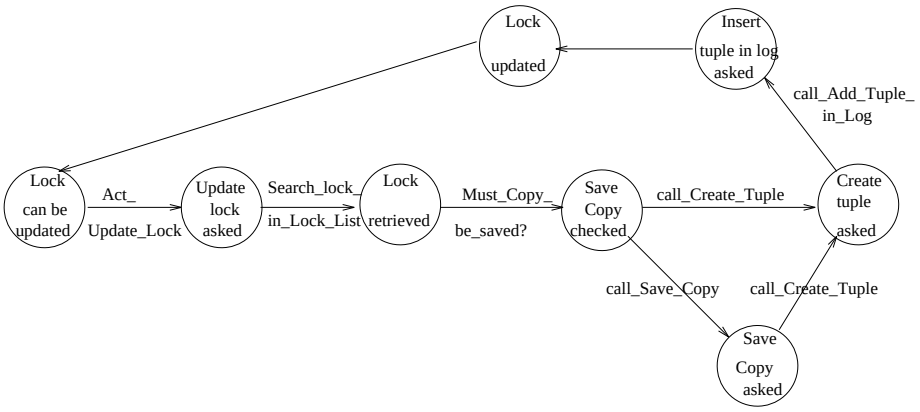


FIGURE 5.4.22. int-Update_Lock: STD of the internal behaviour.

First, the right lock in the Lock_List has to be found. Then it is checked whether a copy has to be saved. A copy will only be saved, if the lock is on a contents of a document and the contents is changed. After that, a tuple is created with the information of the lock and then it is inserted in the log.

This concludes the internal behaviour of the class Process Transaction and its subclasses. Now the operations of the class Process Engine will be described.
The names of the operations are:

Start_WC
Stop_WC
Start_Act
Stop_Act
Refresh

The first operation of class Process Engine is:

Start_WC (User_Name, Role)

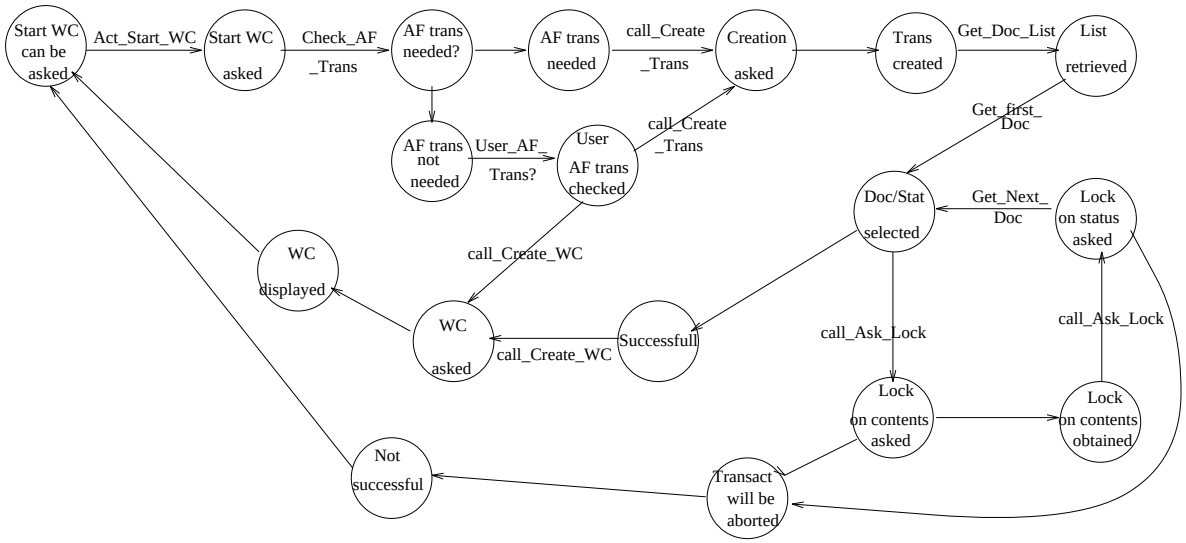


FIGURE 5.4.23. int-Start_WC: STD of the internal behaviour.

A working context has to be created. A working context can be build with Pess_AF protection. It is checked, whether the process model prescribes, that a working context with Pess_AF protection has to be build. If not, the user can choose, whether he wants such a protection.
If a Pess_AF transaction has to be initiated, the transaction is created and the locks are requested. Both contents and status of all document of the working context will be locked. If the lock requests were successful or no pessimistic protection was required, the working context is created.

The second operation of class Process Engine is:

Stop_WC (User_Name, Role)

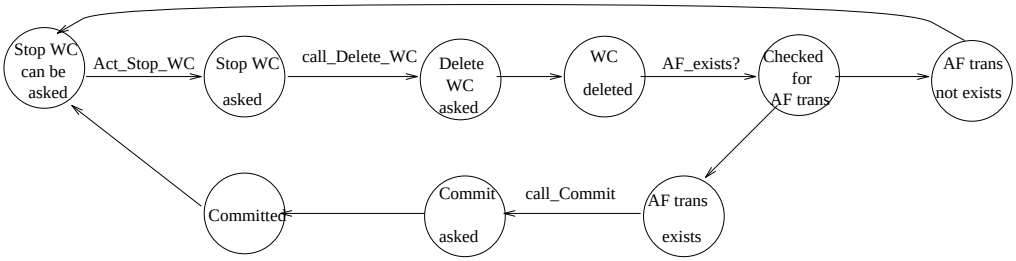


FIGURE 5.4.24. int-Stop_WC: STD of the internal behaviour.

The working context has to be removed. This means that the working context window is deleted and when the working context is build with Pess_AF protection, the Pess_AF transaction will be committed.

The third operation of class Process Engine is:

Start_Act (User_Name, Role, Doc_Name, Activity)

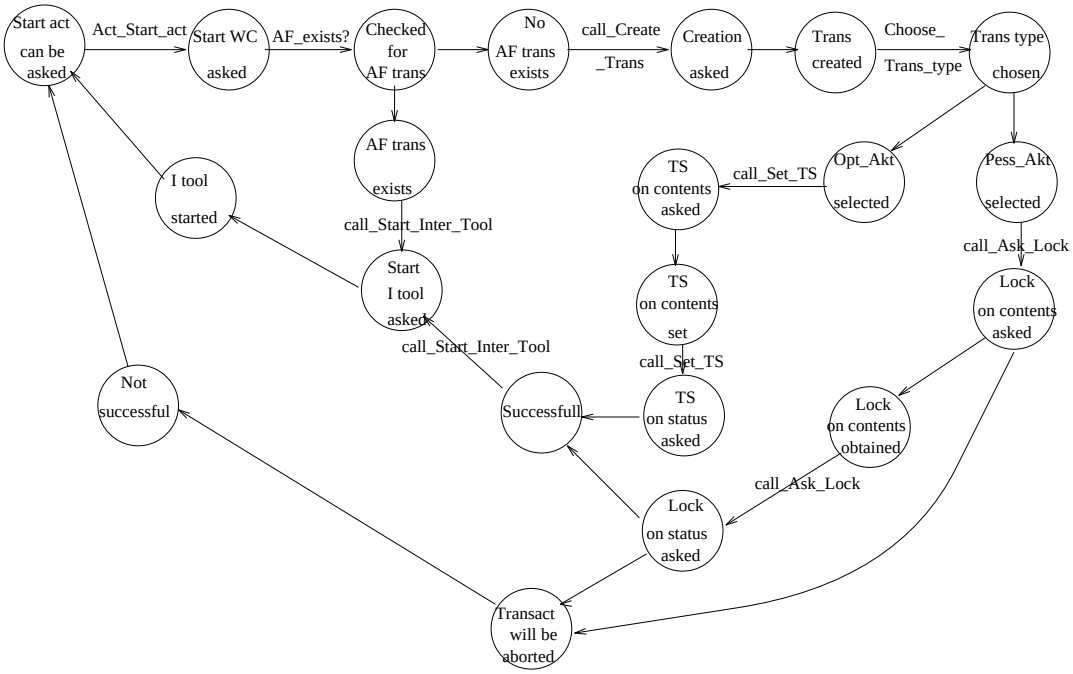


FIGURE 5.4.25. int-Start_Act: STD of the internal behaviour.

An activity may be started. If the working context is build under Pess_AF protection, the interactive tool will be immediately started.
Otherwise, there is decided under what kind of protection the activity will be started (optimistic or pessimistic protection). Now, a transaction will be created and locks or time stamps will be requested. For both the status and the contents of a document, a time stamp or lock will be requested. If locks or time stamps are obtained, the interactive tool will be started. If a requested lock won’t be obtained, the transaction will be aborted and the activity can not be executed.

The fourth operation of class Process Engine is:

Stop_Act (User_Name, Role, Doc_Name, Activity)

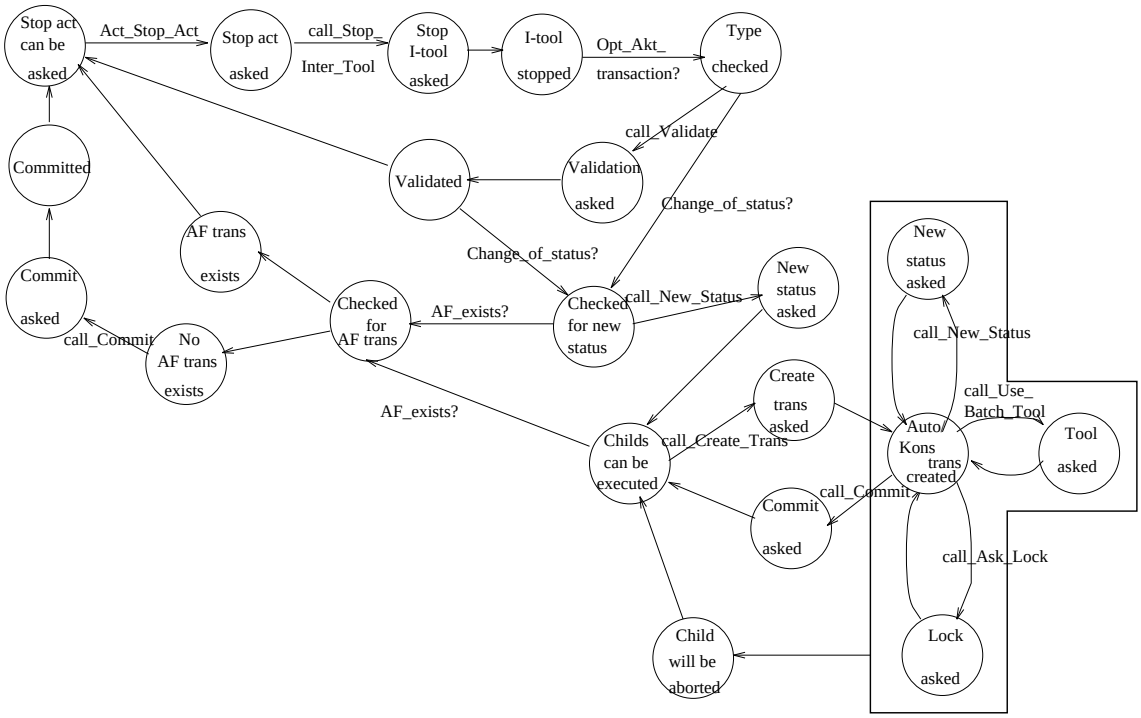


FIGURE 5.4.26. int-Stop_Act: STD of the internal behaviour.

When an activity has to be stopped, the interactive tool will be stopped first. If the activity has used an optimistic protection, the Opt_Akt transaction has to be validated. A successfully validated transaction and a pessimistic transaction will check, whether the status of the document is changed. A change of status will result in child transactions. This means that Auto and/or Kons transactions will be created and that, with or without the usage of tools, data will be changed.
If the activity used Pess_Akt or Opt_Akt protection, the transaction will be committed, otherwise, while the working context is build with Pess_AF protection, the transaction will not be committed.

The fifth and final operation of class Process Engine is:

Refresh (User_Name, Role)

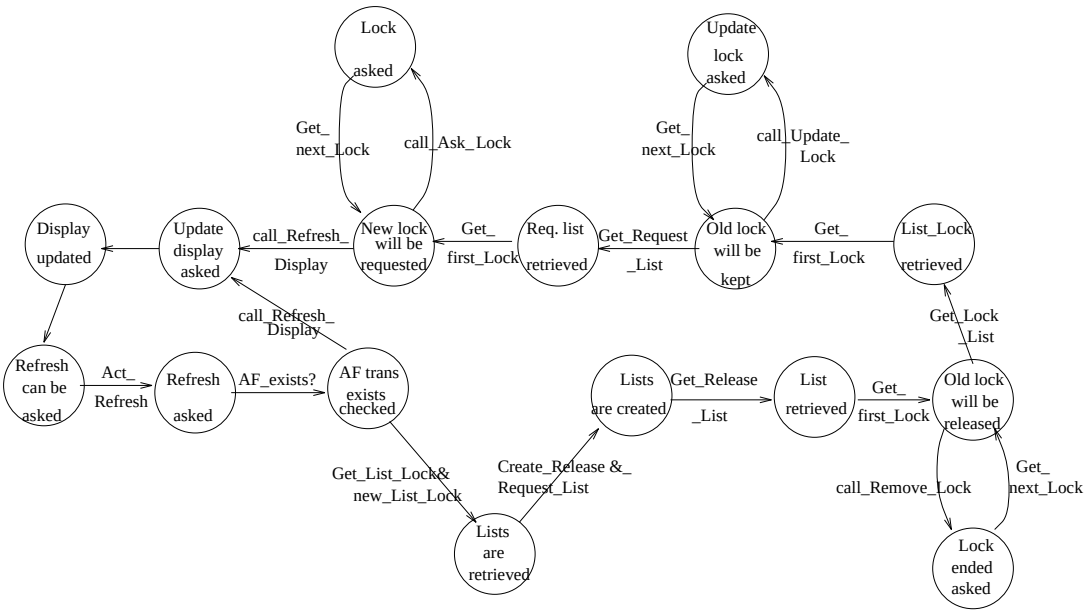


FIGURE 5.4.27. int-Refresh: STD of the internal behaviour.

It is first checked whether the working context is build with pessimistic protection. If not, only the display will be refreshed.
Otherwise, lists will be made. In the Release_List, the locks which have to be released are listed. In Request_List, the locks which have to be requested are listed.
Now, all the locks will be released, that are no longer part of the working context. The information in these locks will be saved and the log will be updated
Then the attribute Lock_List is used to update all the documents, having a lock.
After that, new locks can be requested, if there are needed any. At the end, the display will be refreshed.

This concludes the internal behaviours of the operations of class Process Engine. Next, the operations of the class Contents are specified. All the operations of class Contents are:

Start_Inter_Tool
Stop_Inter_Tool
Use_Batch_Tool
Save_Copy
Create_Copy
Copy_in_Privat

The first operation of class Contents is:

Start_Inter_Tool (Ident)

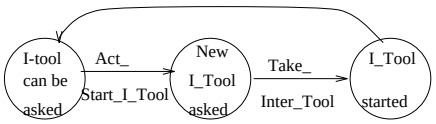


FIGURE 5.4.28. int-Start_Inter_Tool: STD of the internal behaviour.

An interactive tool will be taken, to work on the contents of a document.

The second operation of class Contents is:

Stop_Inter_Tool (Ident)

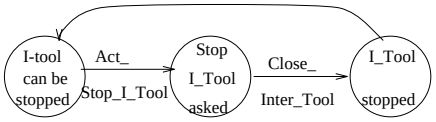


FIGURE 5.4.29. int-Stop_Inter_Tool: STD of the internal behaviour.

The interactive tool will be closed.

The third operation of class Contents is:

Use_Batch_Tool (Ident)

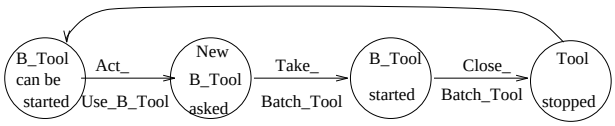


FIGURE 5.4.30. int-Use_Batch_Tool: STD of the internal behaviour.

The batch tool is started, used and eventually it is closed.

The fourth export operation of class Contents is:

Save_Copy

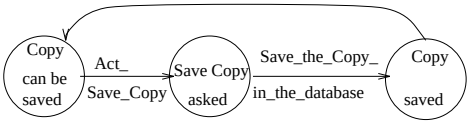


FIGURE 5.4.31. int-Save_Copy: STD of the internal behaviour.

The copy of the contents of a document is saved back in the data base.

The fifth export operation of class Contents is:

Create_Copy

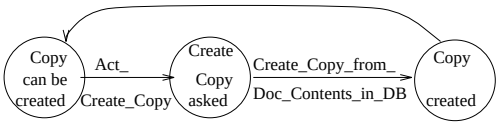


FIGURE 5.4.32. int-Create_Copy: STD of the internal behaviour.

Of the contents of a document, a copy is made. The process engine works on this copy, instead of working directly on the original in the data base.

The sixth and last export operation of class Contents is:

Copy_in_Privat

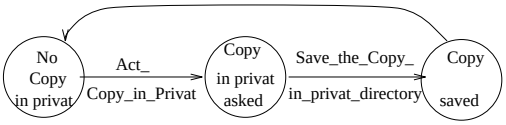


FIGURE 5.4.33. int-Copy_in_Privat: STD of the internal behaviour.

The contents of this document may not be saved back in the data base. The copy will be stored in a private directory of the engineer.

The next class is Status. This class has one export operation.

New_Status (Doc_Status)

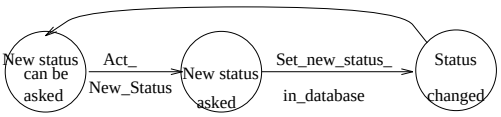


FIGURE 5.4.34. int-New_Status: STD of the internal behaviour.

The new status value of the document is inserted in the data base.

This concludes the internal behaviours of the operations of the subclasses of Document.
The last class, which is specified, is Working Context. the export operations of this class are:

Create_WC
Delete_WC
Refresh_Display

The first operation of class Working Context is:

Create_WC (User_Name, Role)

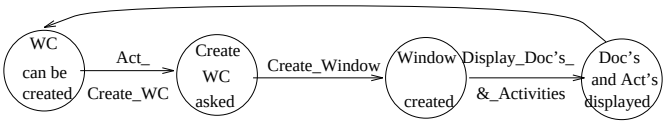


FIGURE 5.4.35. int-Create_WC: STD of the internal behaviour.

The working context window is created and the documents and activities, which the engineer will see, are displayed.

The second operation of class Working Context is:

Delete_WC

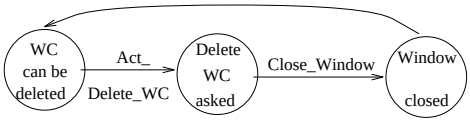


FIGURE 5.4.36. int-Delete_WC: STD of the internal behaviour.

The working context window is closed.

The third and last operation of class Working Contetxt is:

Refresh_Display ()

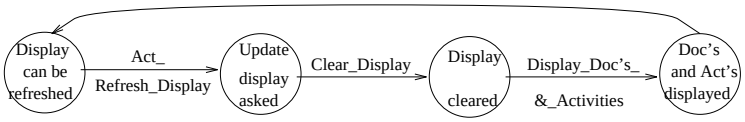


FIGURE 5.4.37. int-Refresh_Display: STD of the internal behaviour.

The display of the already existing working context window will be cleared. Then the new documents and activities are displayed.

5.5 PARADIGM

In the previous sections, the external behaviours of the classes and the internal behaviours of the export operations are given. In this section, the coordination of the internal behaviours with use of the external behaviours is given.
The external behaviours are used as managers, to model this coordination. Each manager manages several employees. The employees are internal behaviours of the same class X as the external behaviour and the internal behaviours, which call operations from this class X.
For each employee, its subprocesses and traps are given.

In this section, from only three classes the managers and all their employees are presented. These classes are Locks, Auto Transaction and Pess_AF Transaction. The presented managers and employees are textually explained pretty extensively. By reading this section, one can understand PARADIGM better, because of the explanation of the diagrams.
Furthermore, in these three managers occur several general problems. These general problems will be the basis of the next chapter. In this chapter, all the problems, which have been identified in these three managers, are listed and solutions are proposed.
In the appendix, the rest of the PARADIGM part, without any textual explanation, can be found.

If one doesn't understand PARADIGM (completely), it would be wise to read the Socca chapter. The following PARADIGM section is nevertheless not that difficult to understand, because of extensive textual explanation of the diagrams.
If one wants to know more of the backgrounds of PARADIGM, one can read [Gr 91]. In [EG 93], another PARADIGM example in Socca is presented,

5.5.1 The manager Locks

The first manager, which is explained, is of the class Locks. In the manager, export operations of class Locks and some export operations of the class Process Transaction or one of its subclasses are coordinated. Some of the export operations of class Process Transaction or one of its subclasses are used, because these operations call operations of the class Locks.

The employees of the manager Locks are:

Get_Read_Lock
Get_Write_Lock
Get_Lock_Father
Release_Lock
Compatibility
Conflict
Ask_Lock
Commit
Remove_Lock
Abort

First, all the internal behaviours of the class Locks are presented.

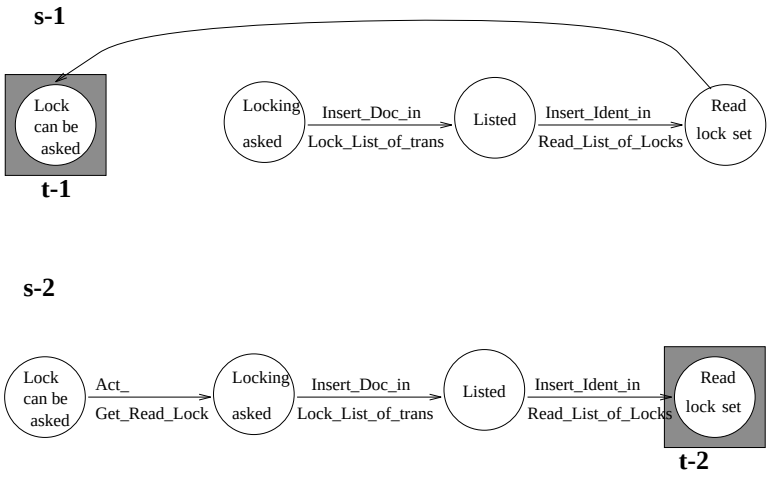


FIGURE 5.5.1.int-Get_Read_Lock's subprocesses and traps w.r.t. Locks.

The internal behaviour Get_Read_Lock is divided in two subprocesses. In subprocess s-1, the internal behaviour can't be activated. If the employee is in subprocess s-1 and trap t-1, the internal behaviour isn't executing. If the employee would switch to subprocess s-2, the employee could leave the state *Lock can be asked* and the internal behaviour can be activated. In trap t-2, the internal behaviour has stopped executing, because the read lock is set.

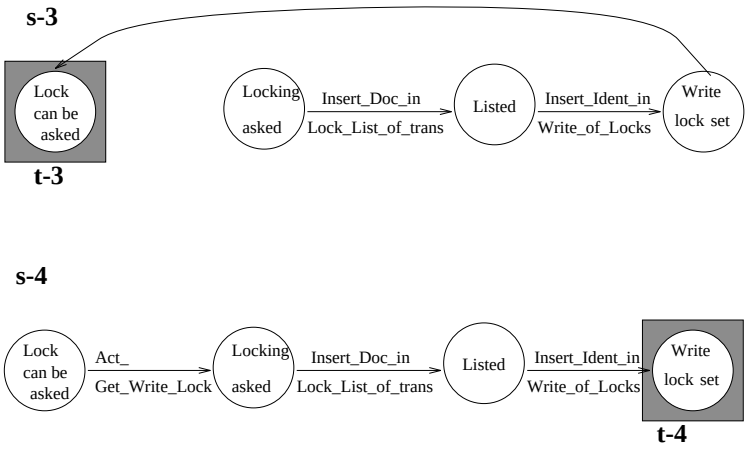


FIGURE 5.5.2.int-Get_Write_Lock's subprocesses and traps w.r.t. Locks.

The subprocesses and traps of this employee and several employees which follow, have similar subprocesses and traps as those from the internal behaviour of Get_Read_Lock. The first subprocess can't activate the internal behaviour and the state before the internal behaviour is activated is a trap. In the second subprocess, the internal behaviour can be activated and the goal of the operation can be reached. The goal of the above employee is to set a write lock. This goal is represented as a trap. Once, the goal is reached, the internal behaviour can't execute any more, unless the employee switches to the other subprocess s-3.
So, in subprocess s-3 and trap t-3, the employee is waiting until it can activate the internal behaviour and in trap t-4 of subprocess s-4 the goal is reached, which means that a write lock is set.

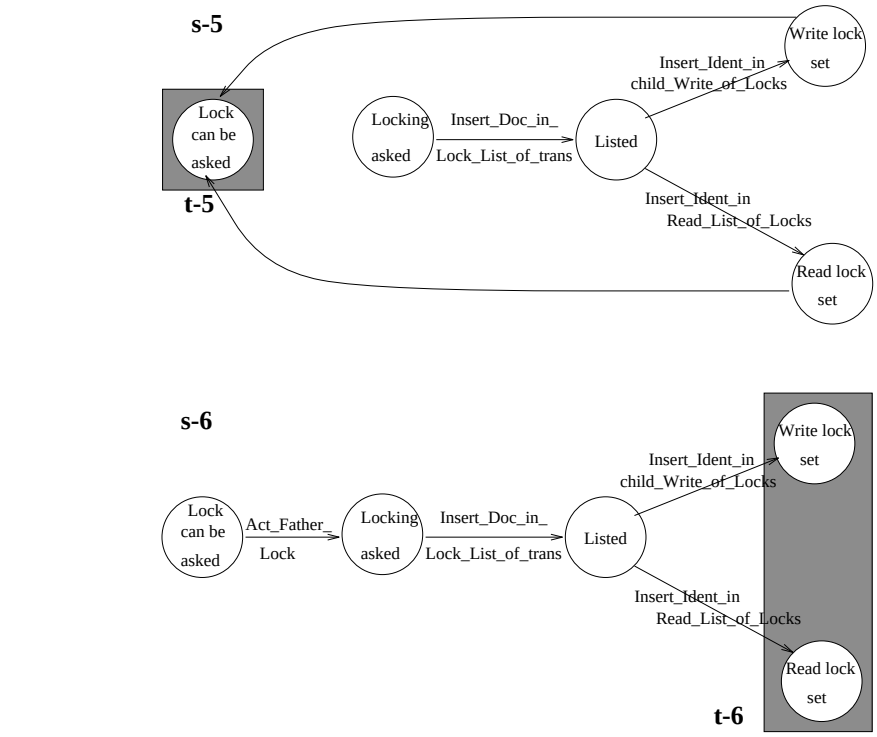
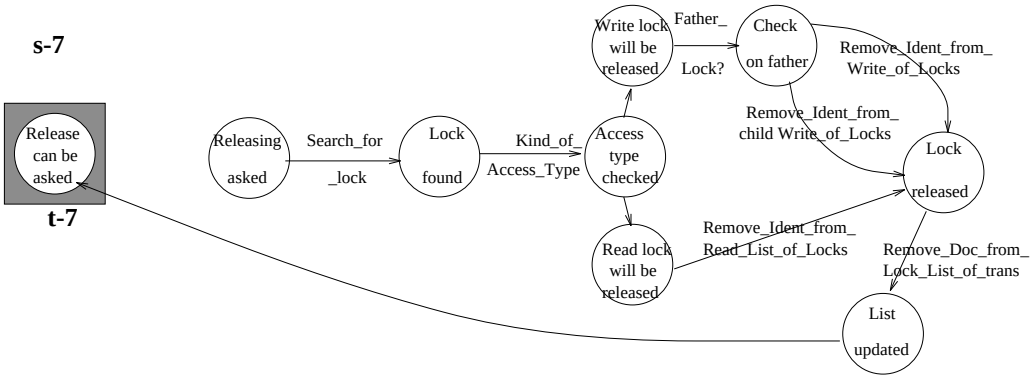


FIGURE 5.5.3.int-Get_Lock_Father's subprocesses and traps w.r.t. Locks.

In trap t-5 of subprocess s-5, the employee is waiting until it can activate the internal behaviour.
In trap t-6 of subprocess s-6, the lock, which is already set by its father, is set. This lock can be a read or write lock. In this subprocess, the trap exists of two states.



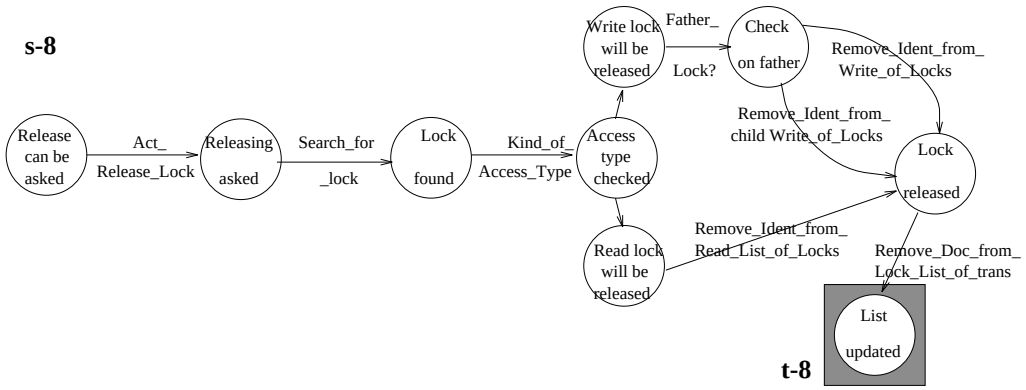


FIGURE 5.5.4.int-Release_Lock's subprocesses and traps w.r.t. Locks.

In trap t-8 of subprocess s-8, the lock is released.

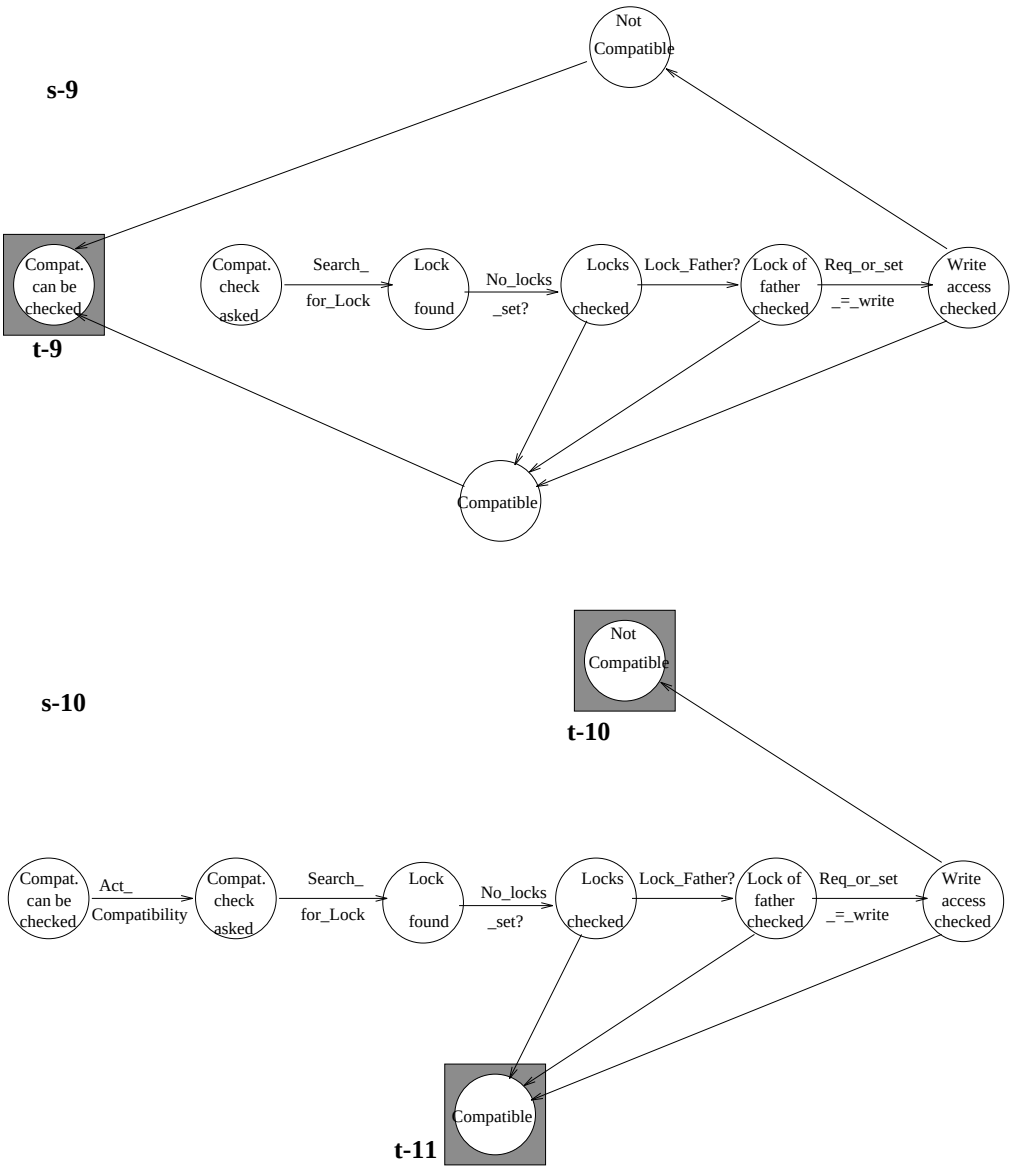


FIGURE 5.5.5.int-Compatibility's subprocesses and traps w.r.t. Locks.

The first subprocess s-9 can’t activate the internal behaviour.
Subprocess s-10 has two traps. So, the operation has two goals.
Either the lock request is compatible with the (possibly) already obtained locks from other transactions and trap t-11 will be reached. Or the lock request is not compatible, which means that the lock request is in conflict with the already obtained locks and trap t-10 will be reached.

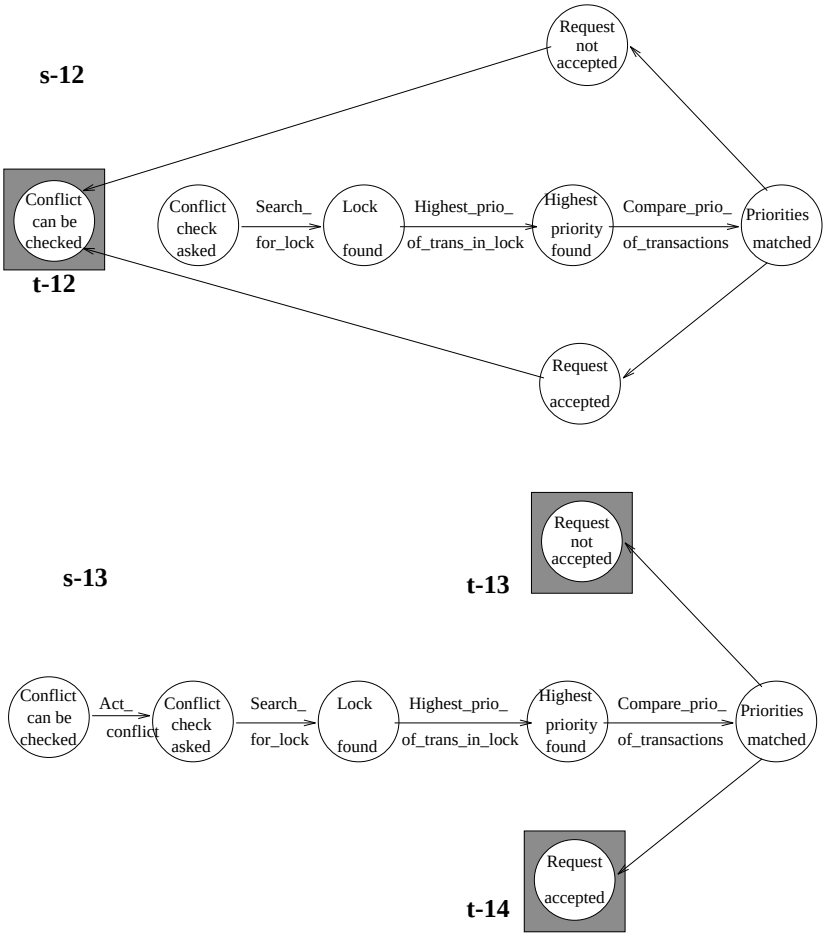


FIGURE 5.5.6.int-Conflict’s subprocesses and traps w.r.t. Locks.

Also here, the second subprocess s-13 has two traps.In trap t-13, the requesting transaction has lost the conflict and in trap t-14, the requesting transaction has won the conflict.

This is the last employee of the class Locks. The next employees are from other classes, calling operations from the class Locks. In this case, all the following employees are from the class Process transaction, or one of its subclasses.

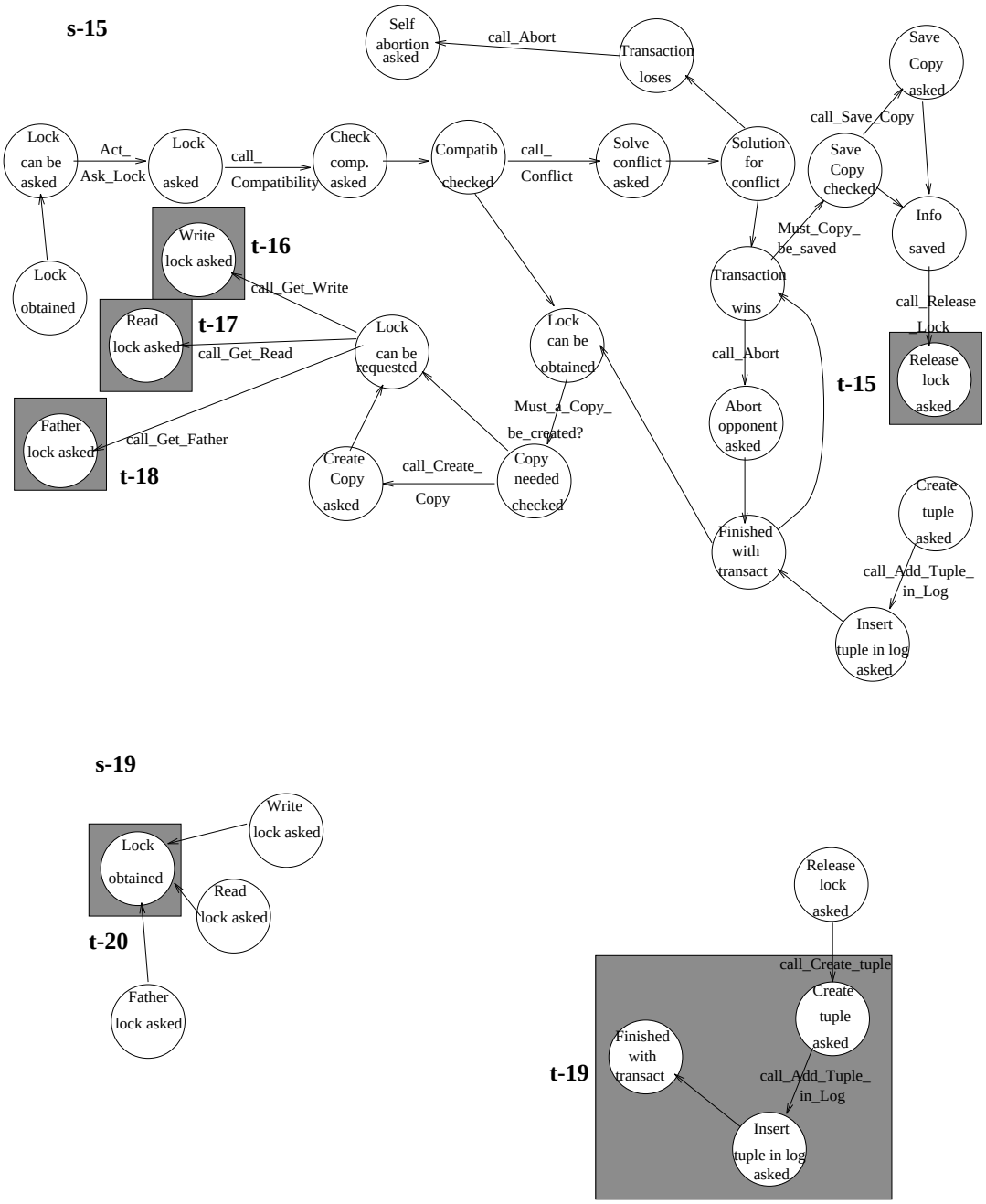


FIGURE 5.5.7.int-Ask_Lock's subprocesses and traps w.r.t. Locks.

In subprocess s-15, four traps can be reached. When the employee is in either one of these traps, it has just called an export operation of the class Locks. This call will result in the activation of the internal behaviour of this operation.
After the call is made and the manager Locks made sure that the internal behaviour of this operation is activated, the employee switches to subprocess s-19. In subprocess s-19, only a couple of states are drawn. The employee may be only in these states after calling the operation. Other states either don't refer to this instance, or can't be reached. The state *Transaction wins*, which can be reached after leaving the state *Finished with transaction*, is referring to another instance. And the state *Info saved* for instance can't be reached any more, after the call to release the lock is made.
If the above internal behaviour has called Release_Lock and the internal behaviour is ended,

the employee may be in one of the states of trap t-19. If the employee is in trap t-20, the internal behaviour, which will set a lock, will end or is ended.

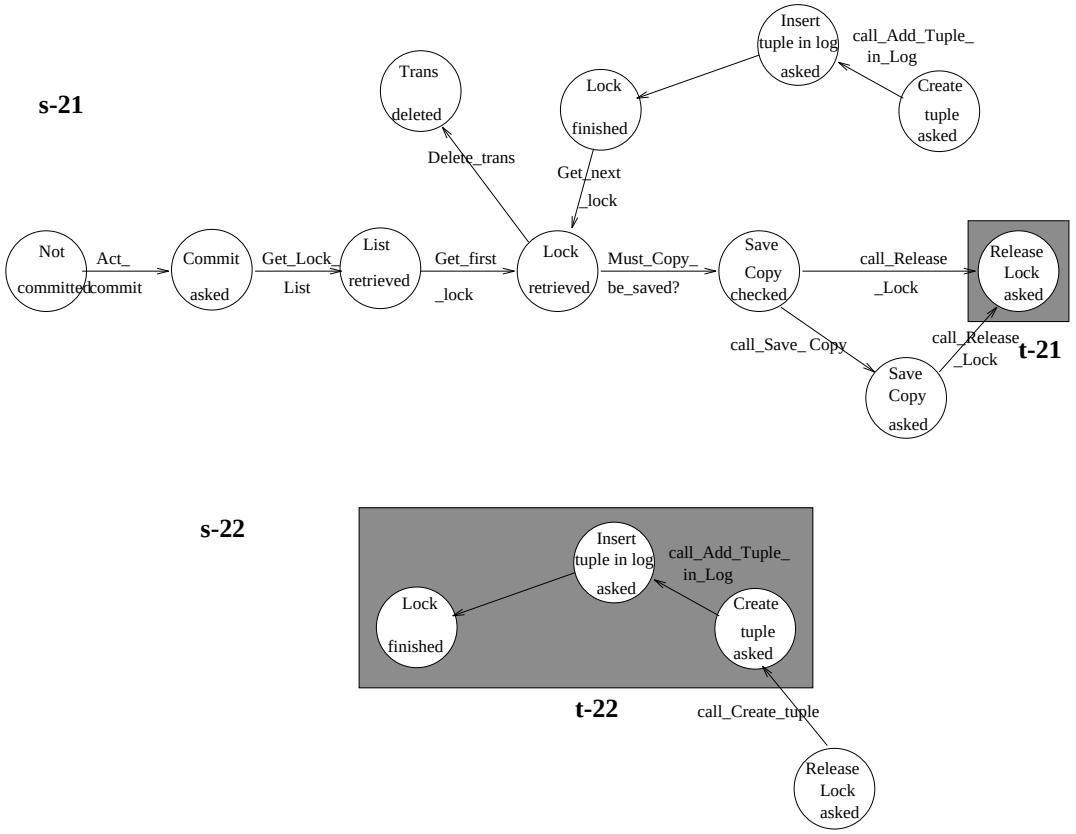
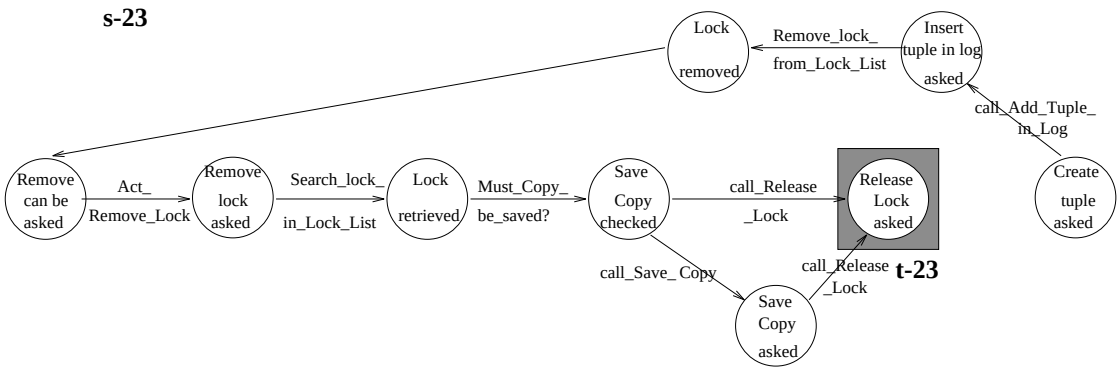


FIGURE 5.5.8.int-Commit's subprocesses and traps w.r.t. Locks.

The choice of a trap in subprocess s-21 is similar to the first subprocess of the internal behaviour Ask_Lock. This trap occurs after the call is made for an export operation of the class Locks. The choice of a trap in the first subprocess occurs in the same way with the following employees.
After the call is made to release a lock, the employee will switch from subprocess s-21 to s-22. When the internal behaviour Release_Lock is ended, the above employee will or has already reached trap t-22.



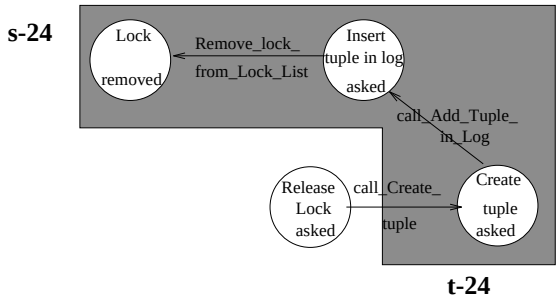


FIGURE 5.5.9.int-Remove_Lock's subprocesses and traps w.r.t. Locks.

In trap t-23 of subprocess s-23, the export operation Release_Lock is called. The internal behaviour of Release_Lock will be activated. Then, the employee switches to subprocess s-24. If the internal behaviour is in trap t-24, the lock will be released or is released.

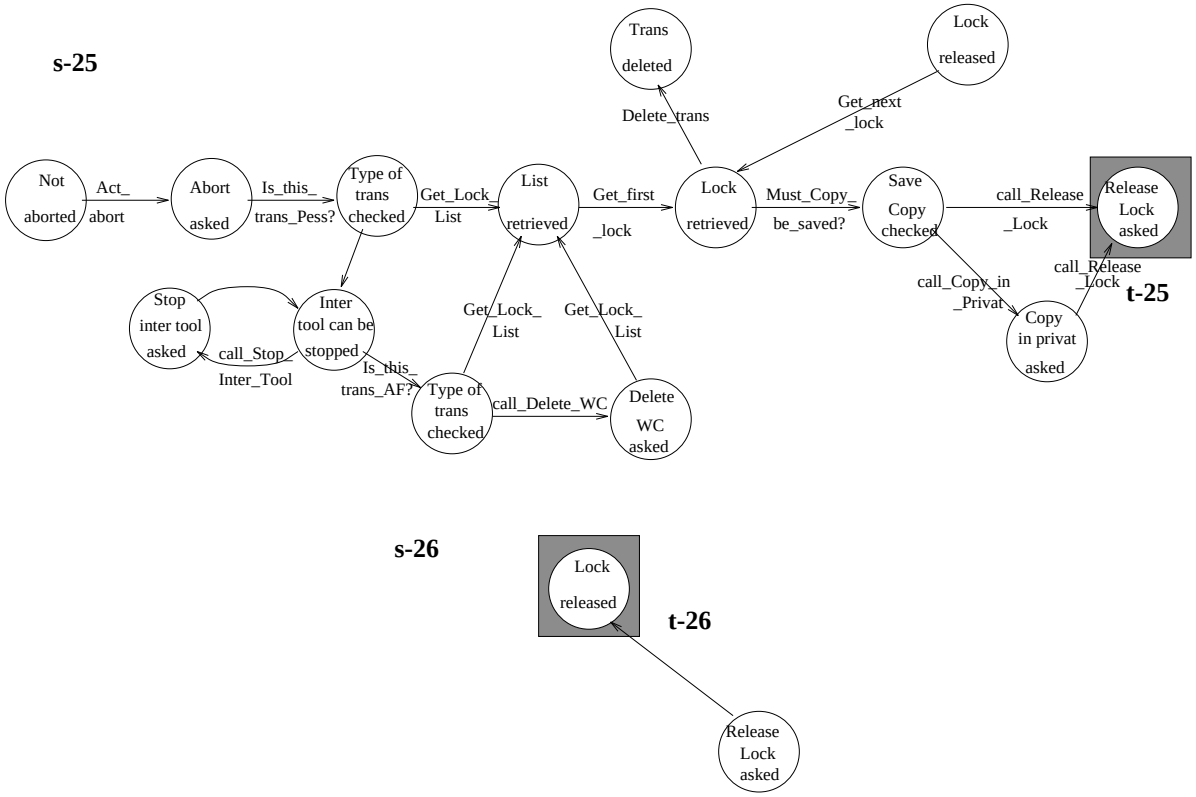


FIGURE 5.5.10.int-Abort's subprocesses and traps w.r.t. Locks.

In trap t-25 of subprocess s-25, the export operation Release_Lock from class Locks is called. When the internal behaviour of Release_Lock can really be activated, the above internal behaviour switches to subprocess s-26. In trap t-26, the operation Release_Lock has released or will release the lock.

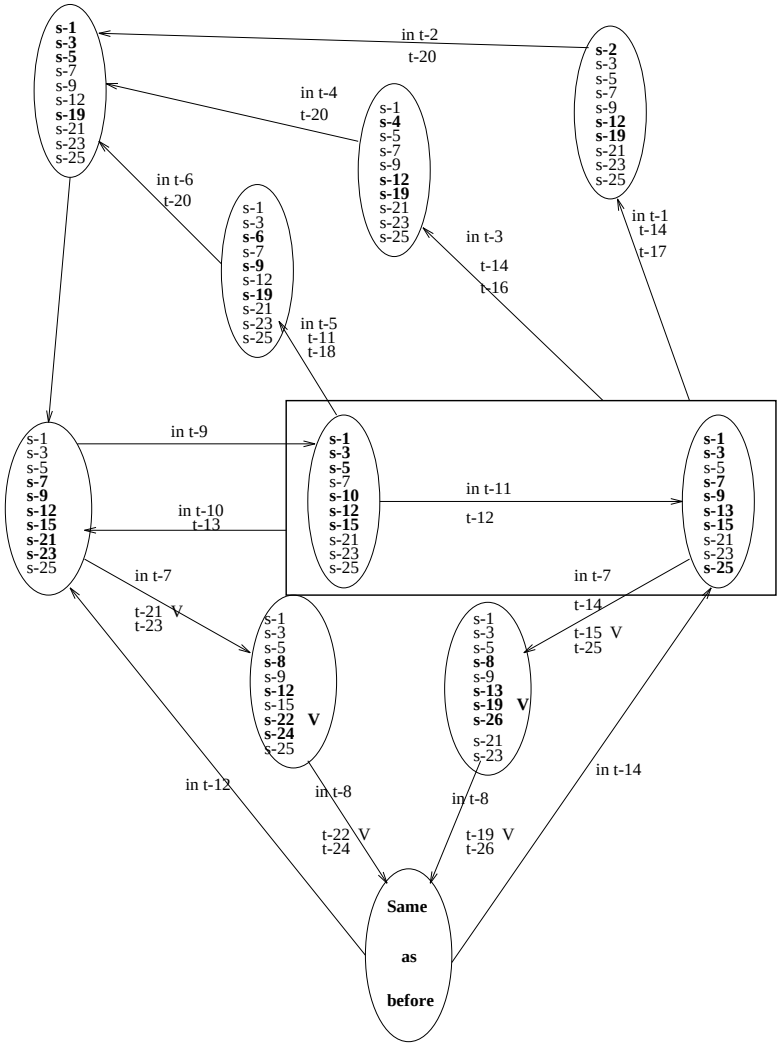


FIGURE 5.5.11. Locks, manager of ten employees.

For the above manager, the external behaviour of Locks is used. In each state of the manager, it is specified in what subprocesses all its employees have to be.

We will focus on the state *Neutral* of the manager, to explain why the employees have to be in a specific subprocess in a state.
The employee Get_Read_Lock has to be in subprocess s-1. The operation can't be activated in the state *Neutral*. The operations Get_Write_Lock and Get_Lock_Father also can't be activated. Therefore, the corresponding employees have to be in subprocess s-3 respectively s-5. Employee Release_Lock has to be in subprocess s-7 in state *Neutral*. The operation can't be activated. If the manager switches to state *Starting to Release*, the employee has to be in trap t-7 and in the new state the employee is switched to subprocess s-8. Operation Compatibility may not be activated in state *Neutral*. The employee is in subprocess s-9. But when the manager wants to go to state *Compatibility checking started*, the employee has to be in trap t-9 and in the new state the employee is switched to subprocess s-10. Operation Conflict can't be activated and therefore the corresponding employee is in subprocess s-12.
Employee Ask_Lock is in state *Neutral* in subprocess s-15. In this subprocess, the employee can trigger the call of an operation of the class Lock. Employee Commit is in state s-21, because it can trigger the call of operation Release_Lock. For the same reason, the employees

Remove_lock and Abort are in trap s-23 respective s-25.

A lot of subprocesses are drawn bold in the states of the manager. This means that the employee, which is drawn bold, can either enter a trap in this state, or has just entered a trap. Employees, which do not execute and therefore can't enter a trap any more (because they are already in a trap) are drawn normal. In this way, it is tried to distinct more clearly between more and less important subprocesses in a specific state.

The sign V occurs several times in the manager. The sign V stands for exclusive or. In the above manager, it is for instance used when from the state *Neutral* the operation Release_Lock is started. Either the employee Commit has to be in trap t-21 or the employee Remove_Lock has to be in trap t-23. This means that only one of the operations call Release_Lock. In state Starting to Release, only one of the two operations will switch from subprocess, because of course only one of them calls the operation Release_Lock.

From two places in the external behaviour, the export operation Release_Lock can be called. The problem is that when both operations are ended (state *Released* is reached), the states in which they enter are not the same. Some subprocesses are different.
The reason for this problem is, that both Release_Lock operations are called from another situation. One Release_Lock is called, because there is a conflict and the other Release_Lock is called, because a process transaction is for instance committing.
The solution, which is chosen for this problem, is to specify that all the subprocesses in state *Released* are the same as in the previous state. To model this is in state *Released* written down 'Same as before'.

It can't be properly modelled in this manager, that the operation Ask_lock of class Process Transaction calls the operation Abort of class Process Transaction. Because the operation Abort is only called by the operation Ask_lock, the operation Abort only calls the right Release_lock operation in the manager. Therefore, the employee Abort can switch from subprocess, when the right Release_Lock operation is called.

5.5.2 The manager Pess_AF Transaction

The next two managers are from class Pess_AF Transaction and Auto Transaction. Both managers have the following three employees:

Create_Trans
Commit
Abort

The above three operations are all inherited from the class Process Transaction and therefore also from the classes Pess_AF Transaction and Auto Transaction.

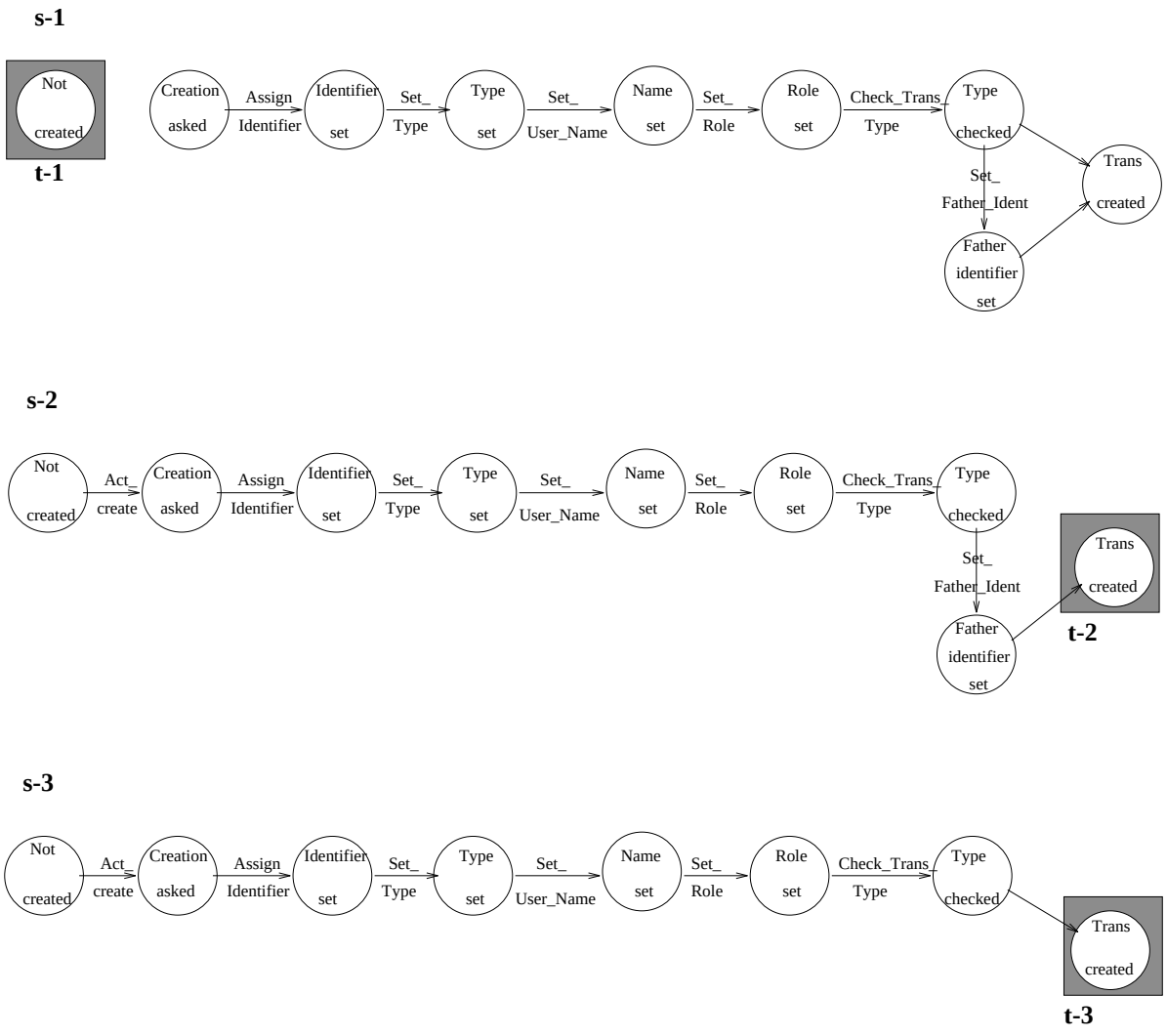


FIGURE 5.5.12. int-Create_Trans's subprocesses and traps w.r.t. Process Transaction.

The employee Create_Trans has three subprocesses. In subprocess s-1 and trap t-1, the behaviour can't be activated. By switching to either subprocess s-2 or subprocess s-3, the behaviour can be activated.
The behaviour switches to subprocess s-2 when the transaction, which will be created, is of the type Auto or Kons and therefore a so-called child transaction.

The behaviour switches to subprocess s-3 when the creating transaction will be of the type Pess_AF, Pess_Akt or Opt_Akt.

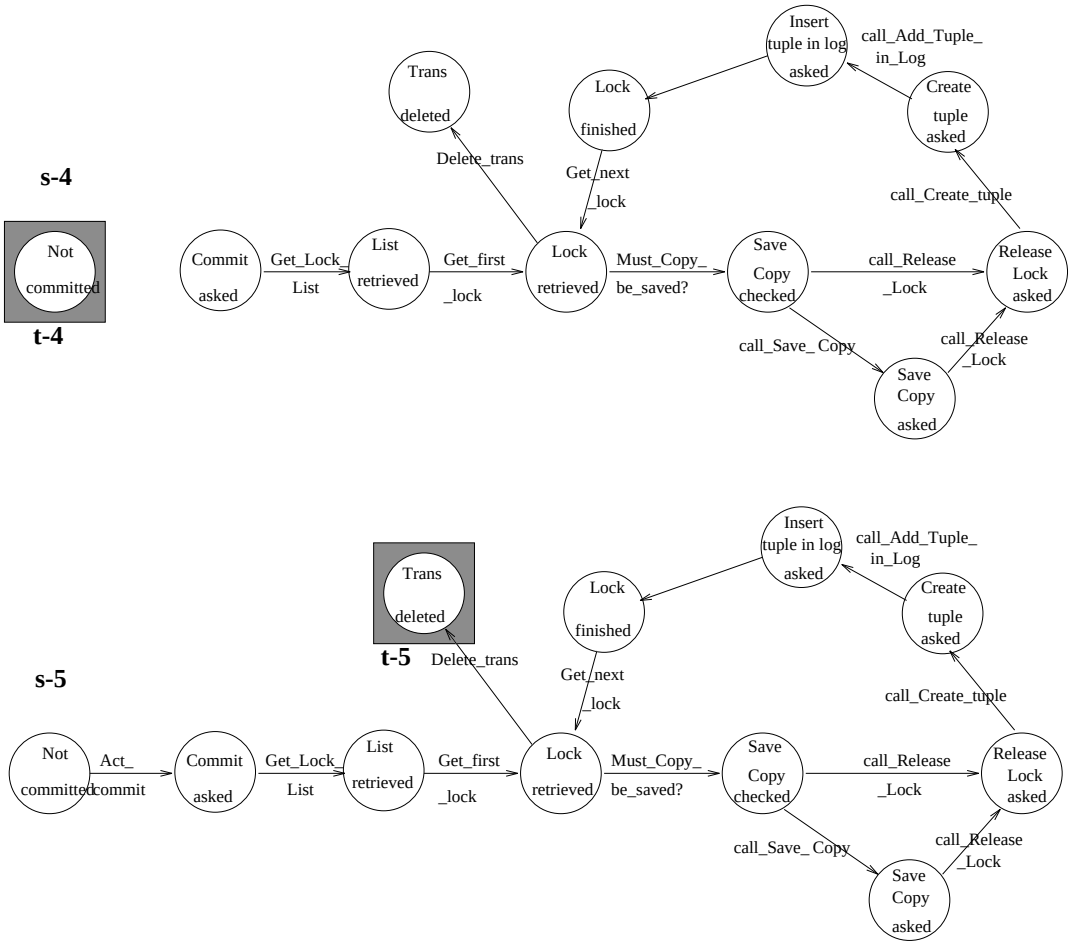
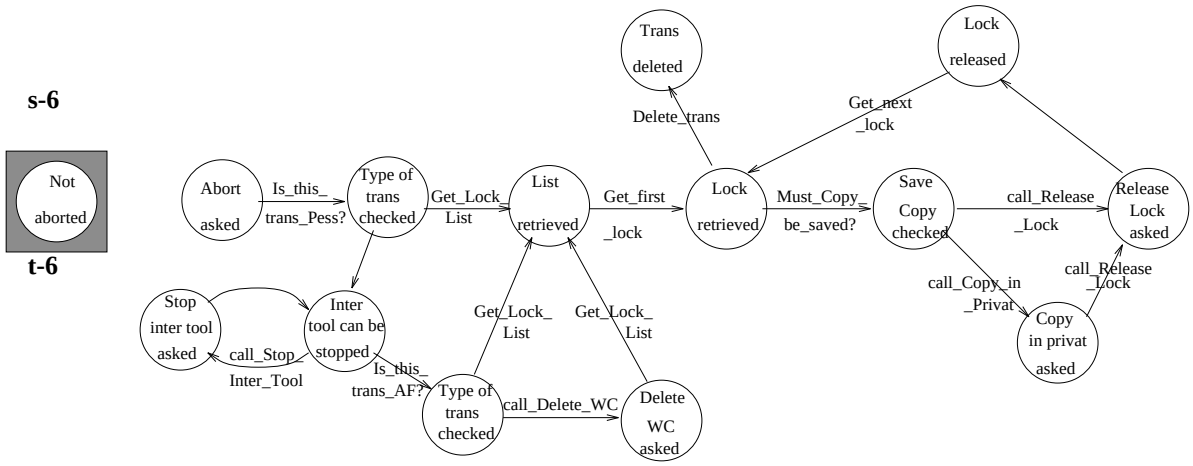


FIGURE 5.5.13. int-Commit's subprocesses and traps w.r.t. Process Transaction.

In subprocess s-4 and trap t-4, the behaviour can't be activated. By switching to subprocess s-5, the behaviour will be activated. If the trap t-5 is reached, the transaction is committed and the operation Commit is finished with executing.



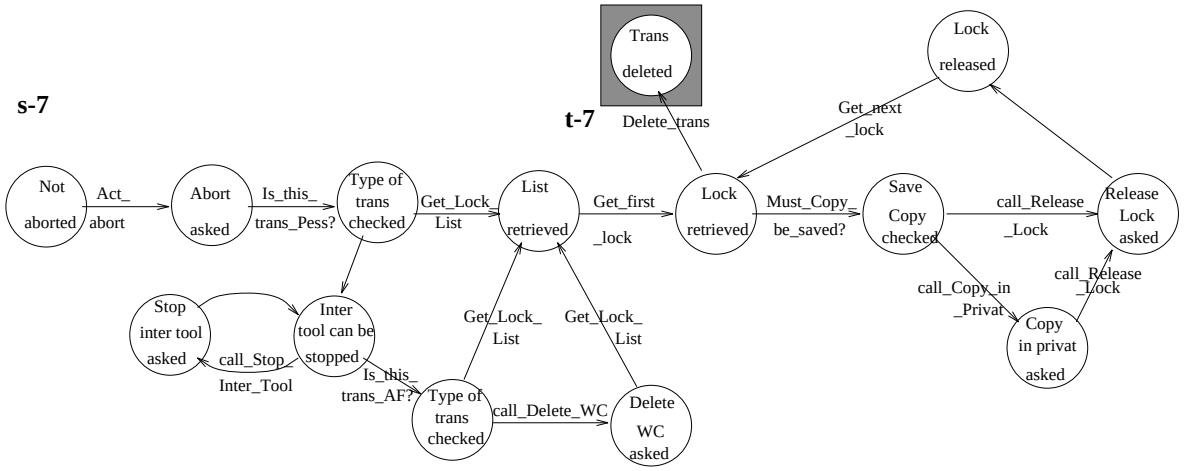


FIGURE 5.5.14. int-Abort's subprocesses and traps w.r.t. Process Transaction.

In subprocess s-6 and trap t-6, the behaviour can't be activated. When the employee switches to subprocess s-7, the behaviour can be activated. In trap t-7, the transaction is aborted.

Besides the three already described employees, the manager Pess_AF Transaction has also some other ones. These employees are:

Ask_Lock (same instance)
Remove_Lock
Update_Lock
Refresh
Start_WC
Stop_WC
Ask_Lock (other instance)

First, the employees from class Pess_AF Transaction will be described.

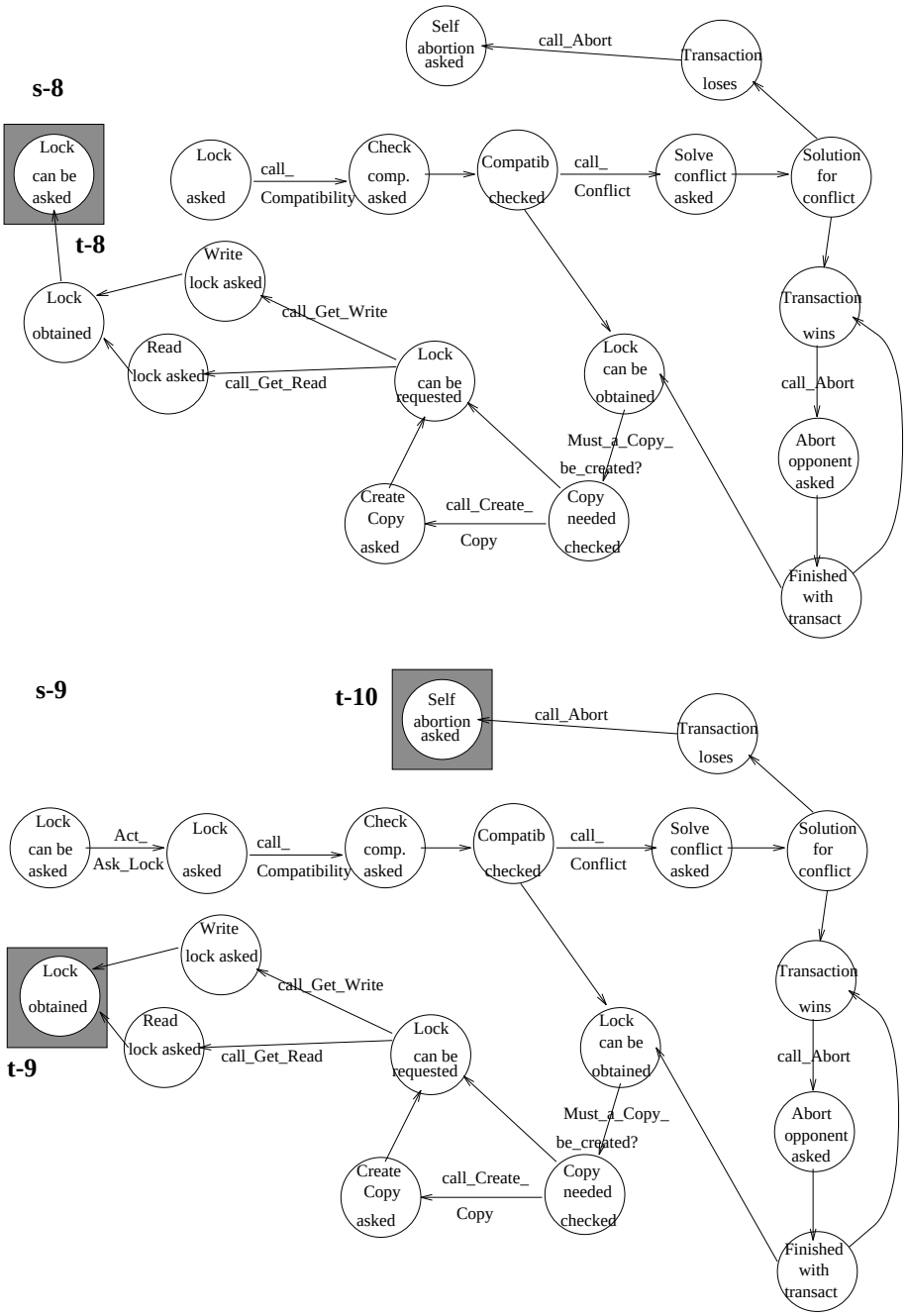


FIGURE 5.5.15. int-Ask_Lock's subprocesses and traps w.r.t. Pess_Akt Transaction.

If the subprocess switches to subprocess s-9, the operation will be activated. Either the request for a lock will be rejected and the transaction will be aborted or the transaction will obtain the requested lock.
If the transaction will be aborted, trap t-10 will be reached. When the transaction obtains the lock, trap t-9 will be reached.
The behaviour of the above operation Ask_Lock is partly restricted, when one compares it with the diagram in the internal behaviour chapter. The reason for this restriction is, that a Pess_AF transaction can't release a lock from another transaction. It can only abort other transactions.

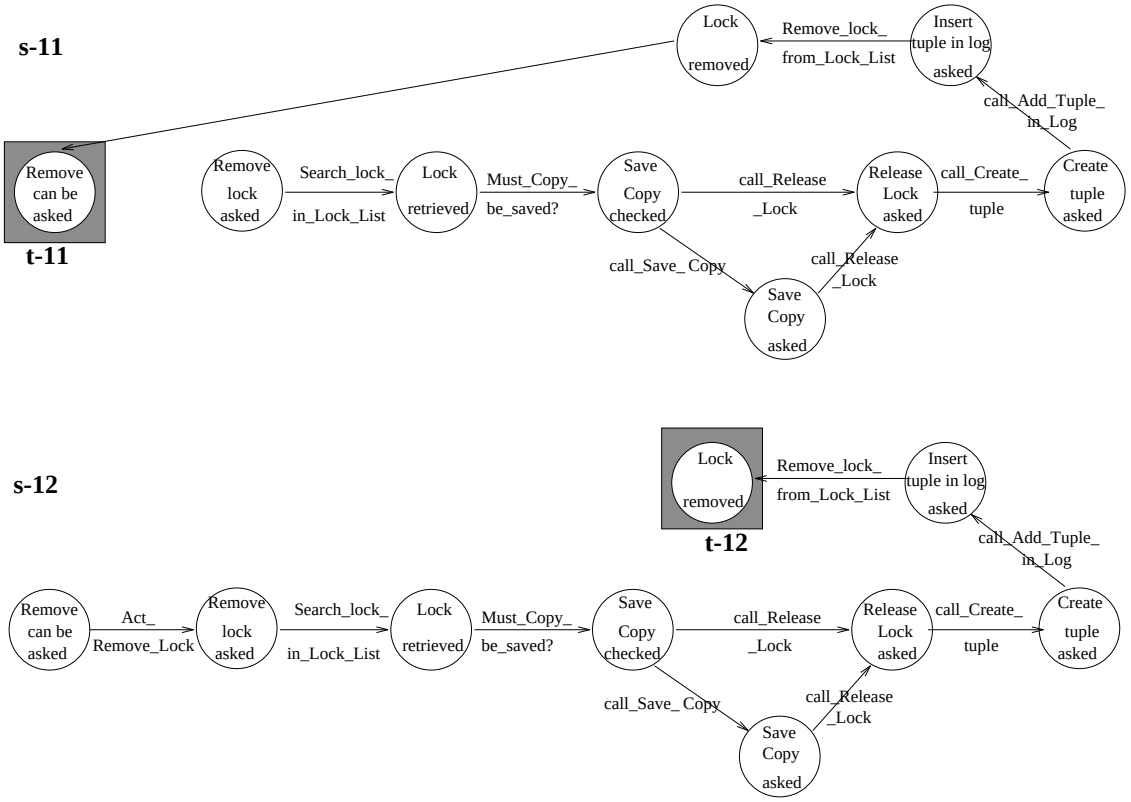


FIGURE 5.5.16. int-Remove_Lock's subprocesses and traps w.r.t. Pess_AF Transaction.

If the subprocess switches to subprocess s-12, the behaviour can be activated and if trap t-12 is reached, then the lock is removed.

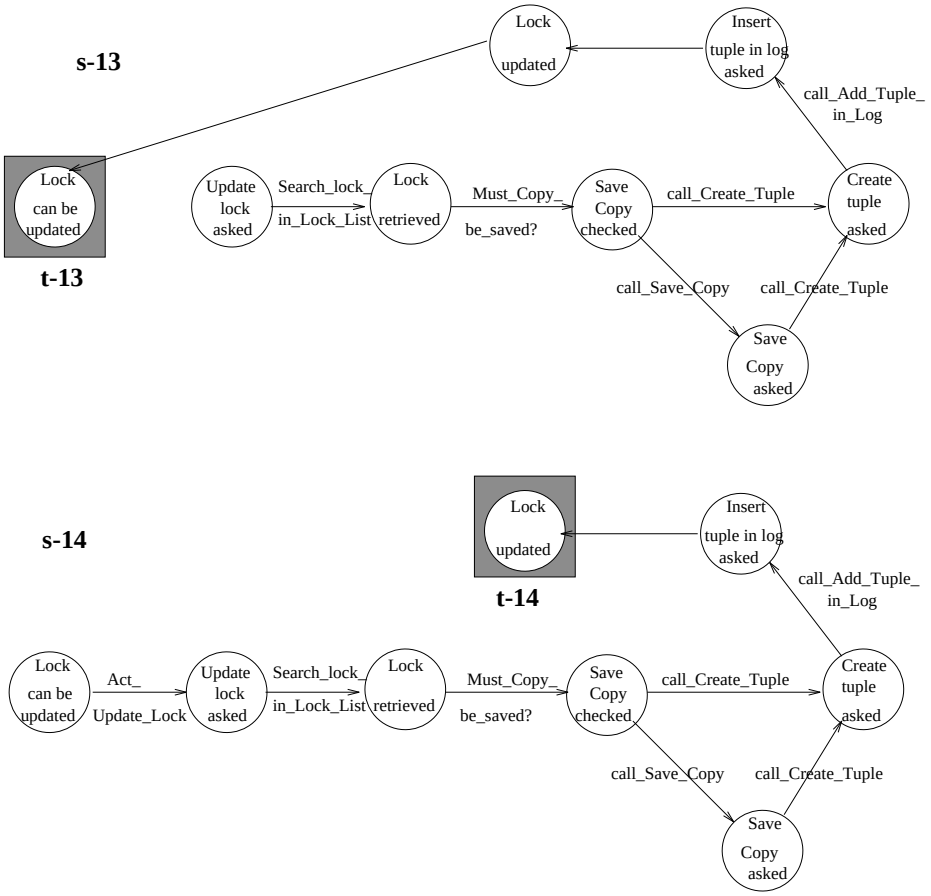
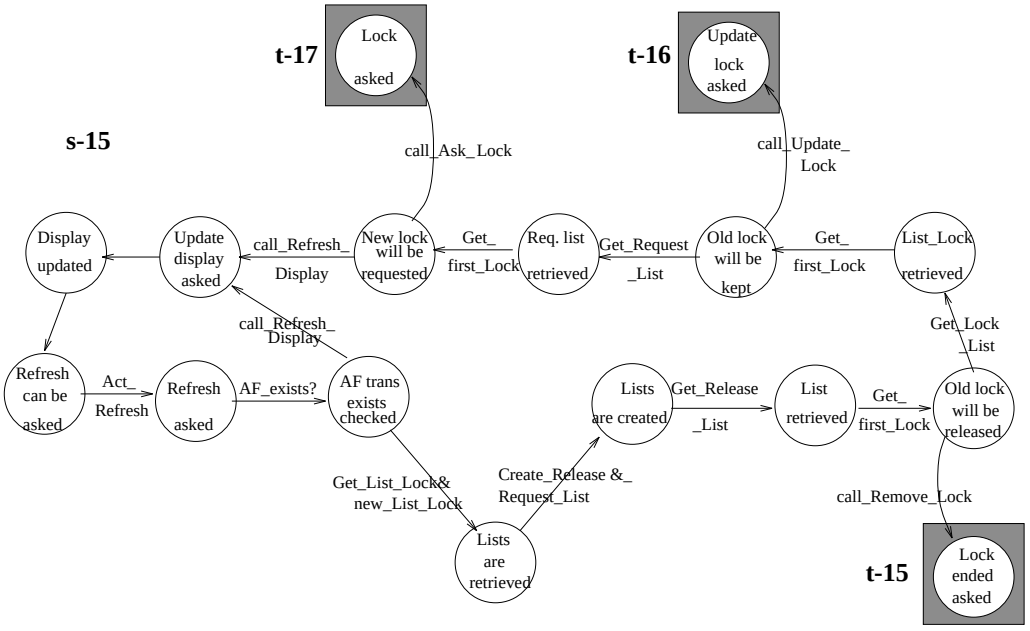


FIGURE 5.5.17. int-Update_Lock's subprocesses and traps w.r.t. Pess_AF Transaction.

If the behaviour has reached trap t-14 in subprocess s-14, the lock has been updated.

The next employees are internal behaviours from other classes, calling operations from the class Pess_AF Transaction.



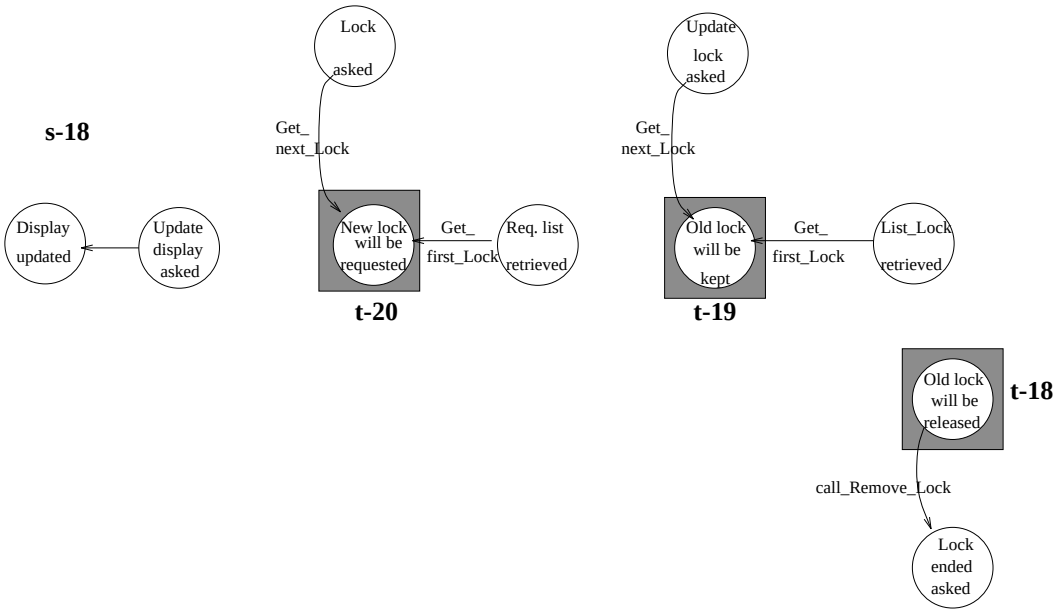
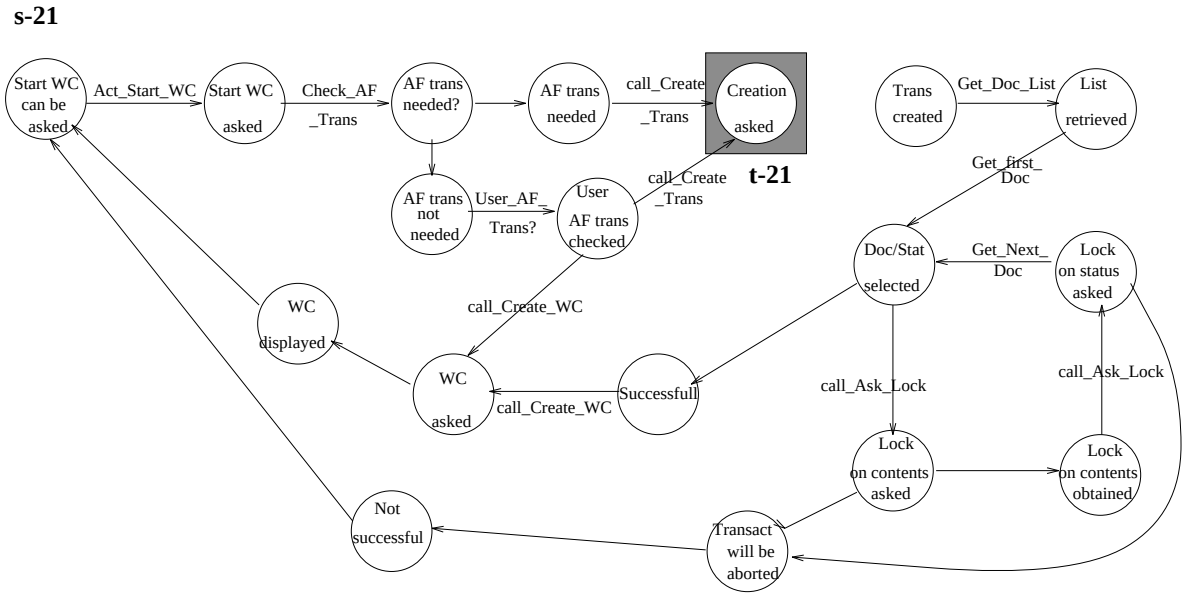


FIGURE 5.5.18. int-Refresh's subprocesses and traps w.r.t. Pess_AF Transaction.

In subprocess s-15, three traps can be entered. When the behaviour is in either one of these traps, an operation from the class Pess_AF Transaction just has been called.
After the call is made, the employee switches to subprocess s-18. If a trap in subprocess s-18 is entered, then the called operation is finished.



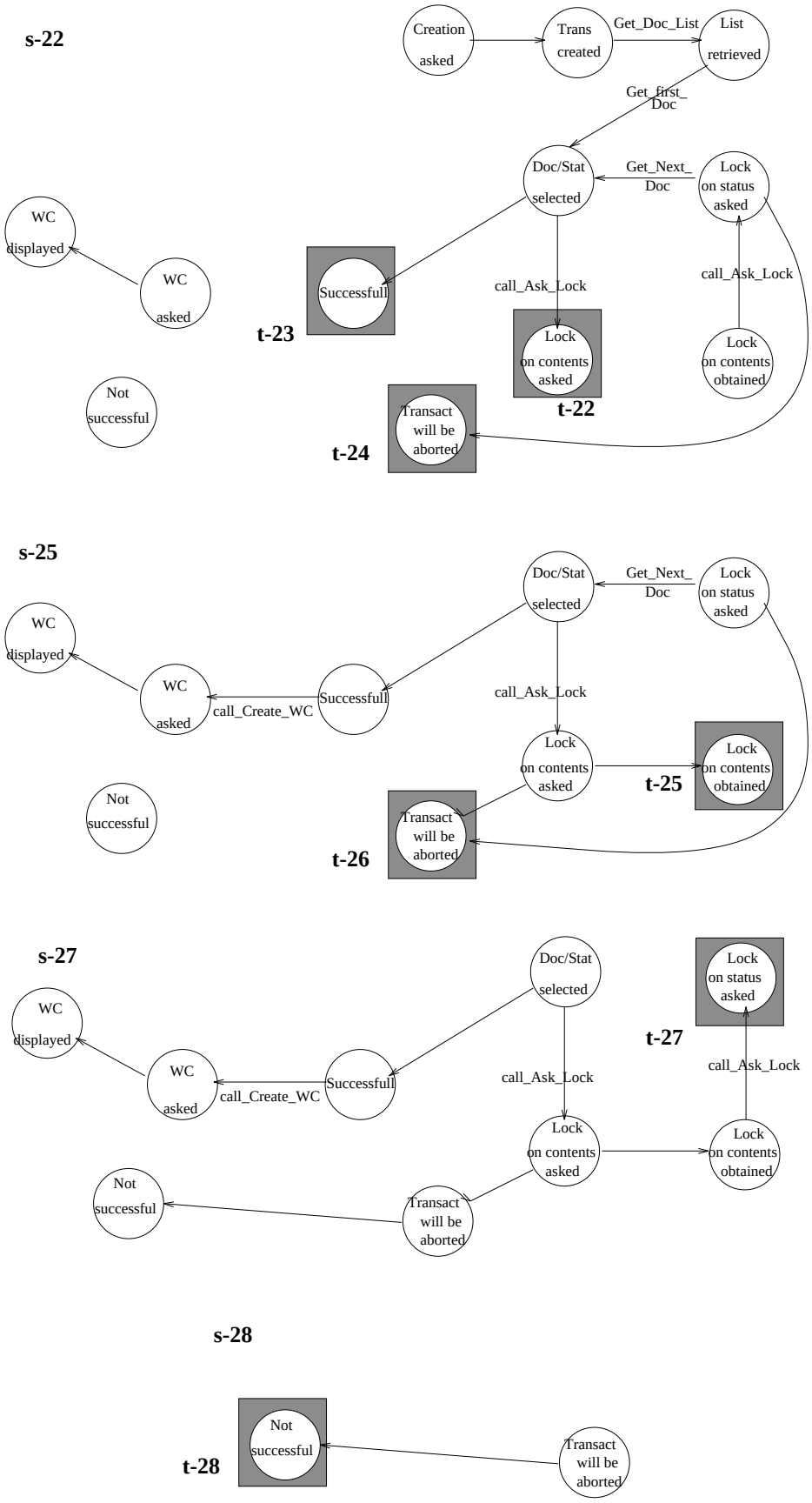


FIGURE 5.5.19. int-Start_WC's subprocesses and traps w.r.t. Pess_AF Transaction.

This employee is divided in five subprocesses.
In trap t-21 of subprocess s-21, an export operation of the class Pess_AF Transaction is called. The transaction will be created. The employee switches to subprocess s-22. If the employee enters a trap, the transaction already has to be created. By entering trap t-22, the operation Ask_Lock is called. In trap t-23, all the locks have been obtained and the transaction will protect the execution on documents. In trap t-24, a requested lock won't be obtained and the transaction will be aborted. The call for this abortion is made somewhere else. This behaviour just reacts on that call. In trap t-28 of subprocess s-28, the transaction is aborted.

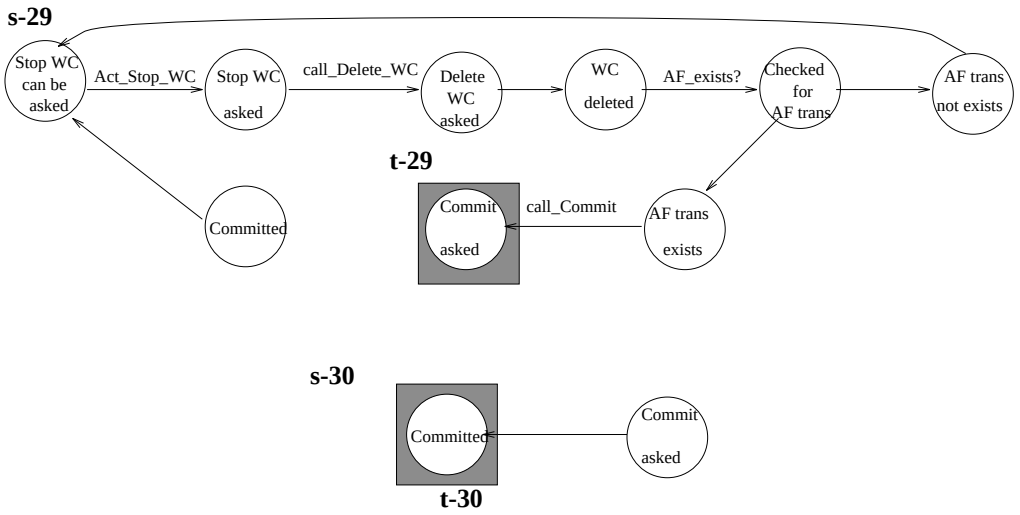
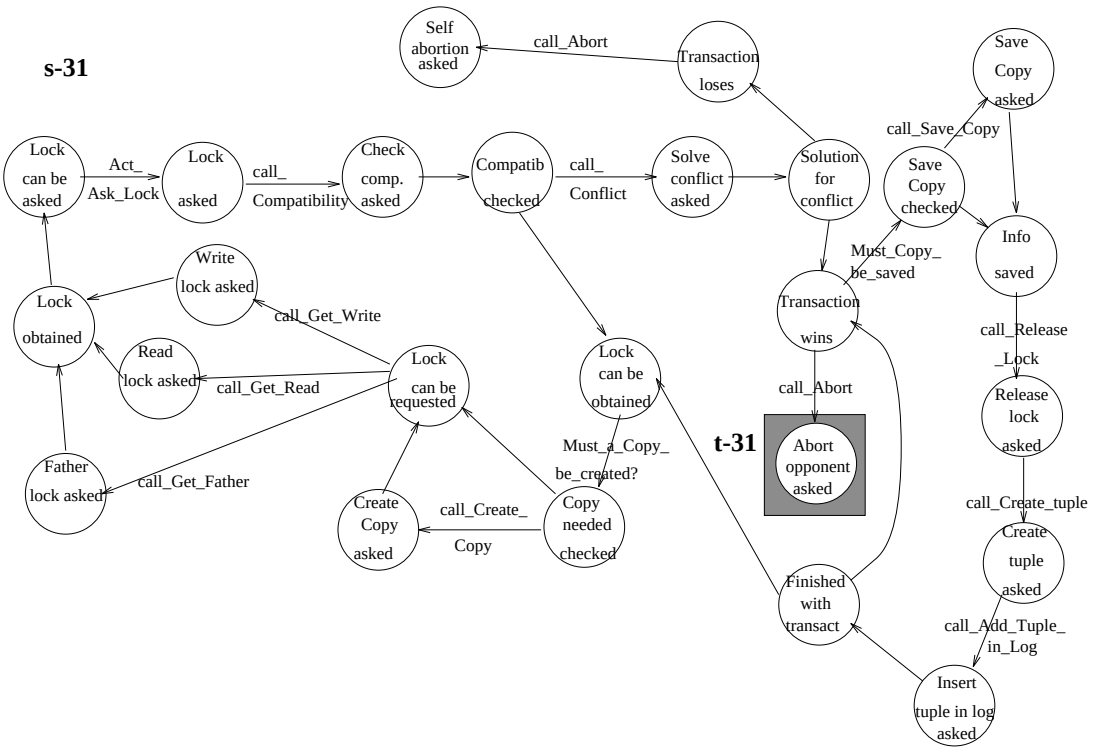


FIGURE 5.5.20. int-Stop_WC's subprocesses and traps w.r.t. Pess_AF Transaction.

The export operation Commit is called. In trap t-30, the transaction is really committed.



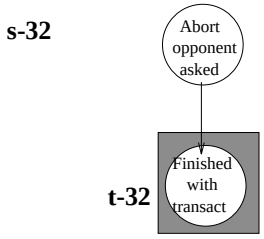


FIGURE 5.5.21. int-Ask_Lock's subprocesses and traps from Kons Transaction w.r.t. Pess_AF Transaction.

The above employee has to be an instance of the class Kons Transaction, because only a Kons transaction can win a conflict from a Pess_AF Transaction. By entering trap t-31, a Kons transaction aborts this Pess_AF transaction.

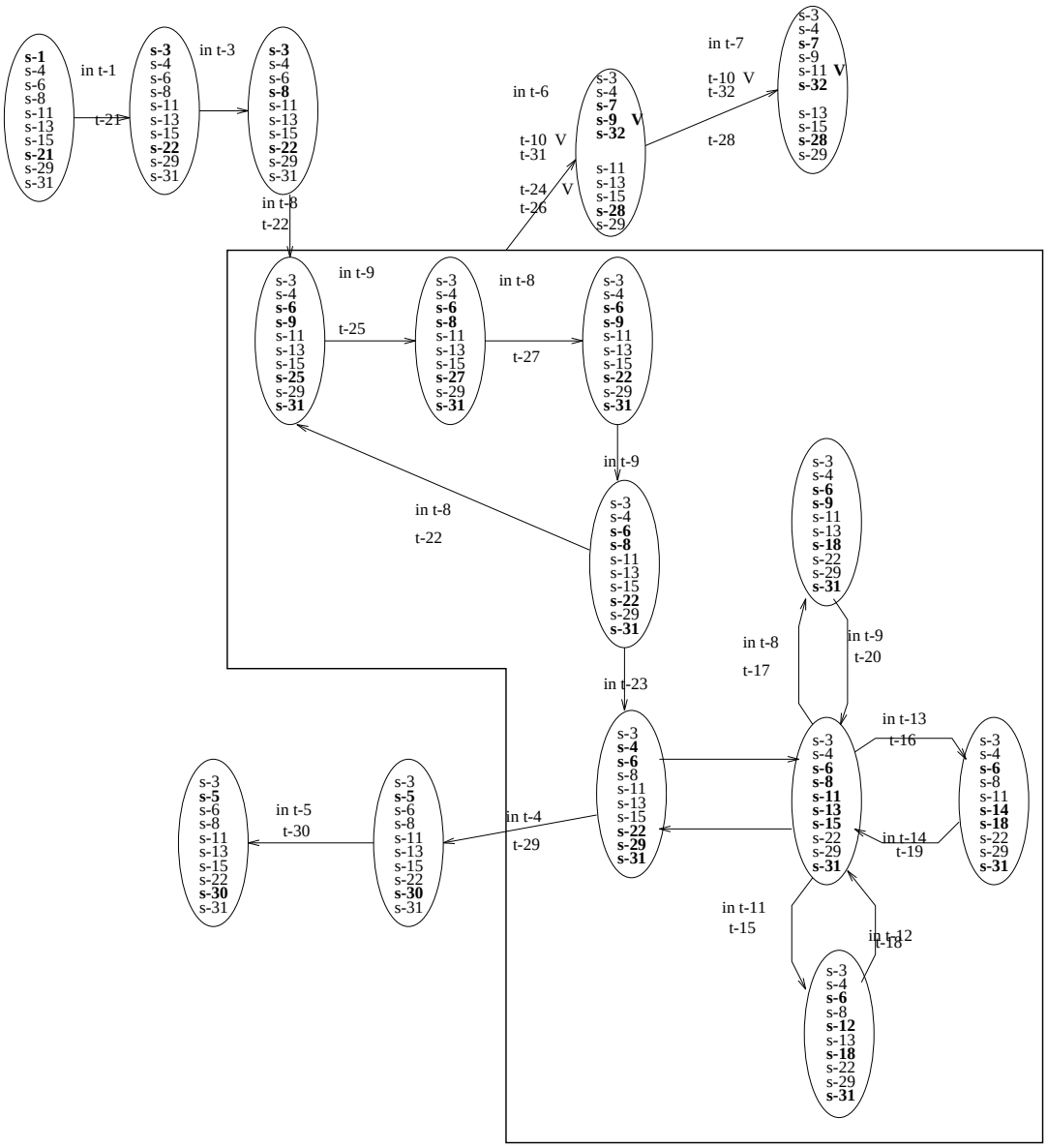


FIGURE 5.5.22. Pess_AF Transaction, manager of ten employees.

For the above manager is of course the external behaviour of Pess_AF Transaction used.

In a lot of operations the operation Commit is for instance called. Only the operation Stop_WC calls the operation Commit of the class Pess_AF Transaction. A lot of operation calls are attribute sensitive. In the above case, it is checked what the value of the attribute Type is.

It is difficult to model, whether a Kons transaction will release a lock or abort the whole transaction. A couple of complex rules make the decision for what will happen. This decision is made in state *Transaction wins* of the operation Ask_Lock.
In figure 21, it is only modelled when the operation aborts a Pess_AF transaction, but it can't be modelled when a lock will be released. It doesn't appear in the external behaviour, when a lock will be released.

In the above manager, some transitions between states aren't labelled with traps. It is difficult to model when the behaviour switches from the state *Execute* to the state *Refresh*.

5.5.3 The manager Auto Transaction

The class Auto transaction has some inherited export operations of class Process Transaction. We refer to 5.5.2. for the specification of subprocesses and traps of these export operations. Besides these three already specified employees, the manager also manages the next three employees.

Ask_Lock (same instance)
Stop_Act
Ask_Lock (other instance)

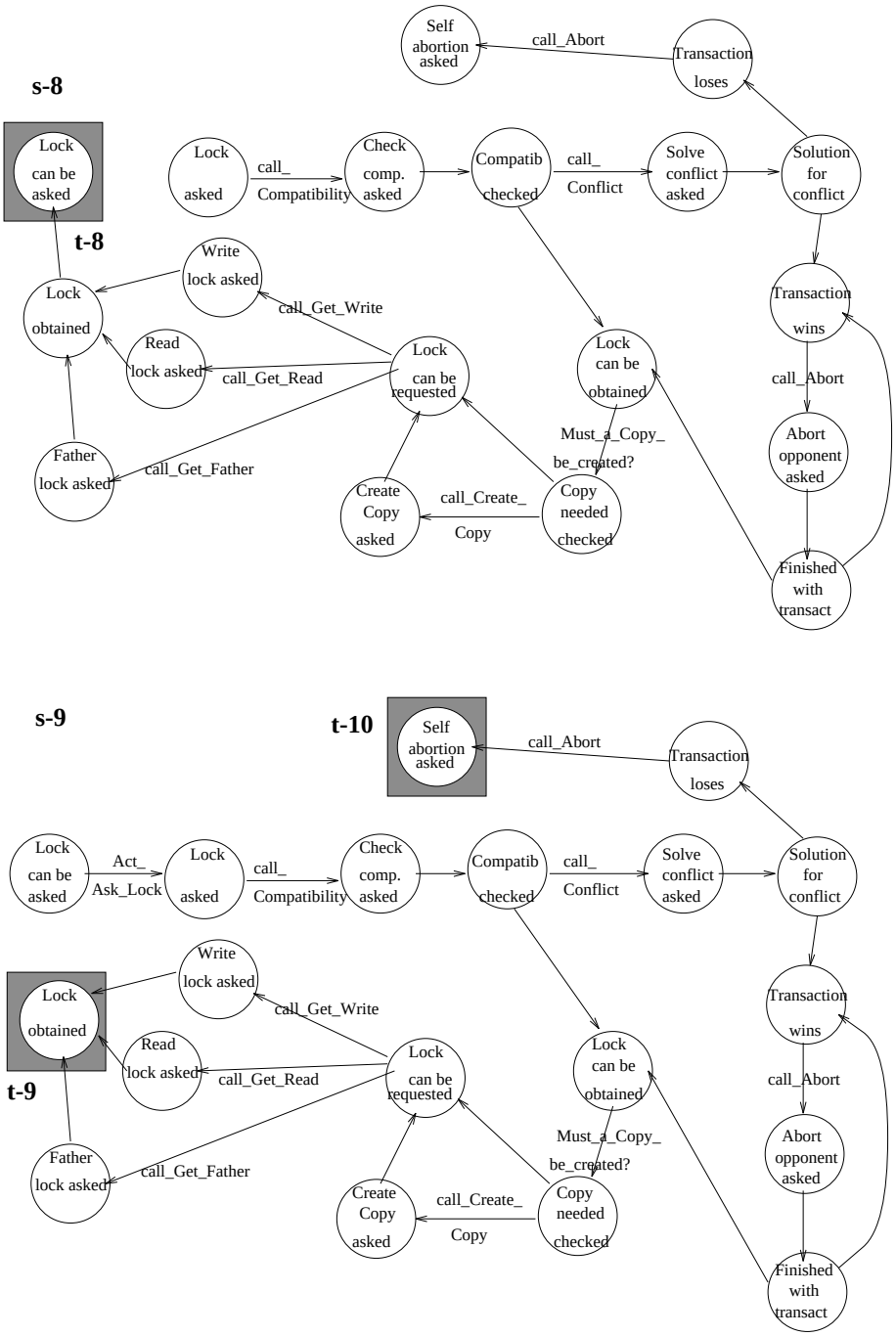
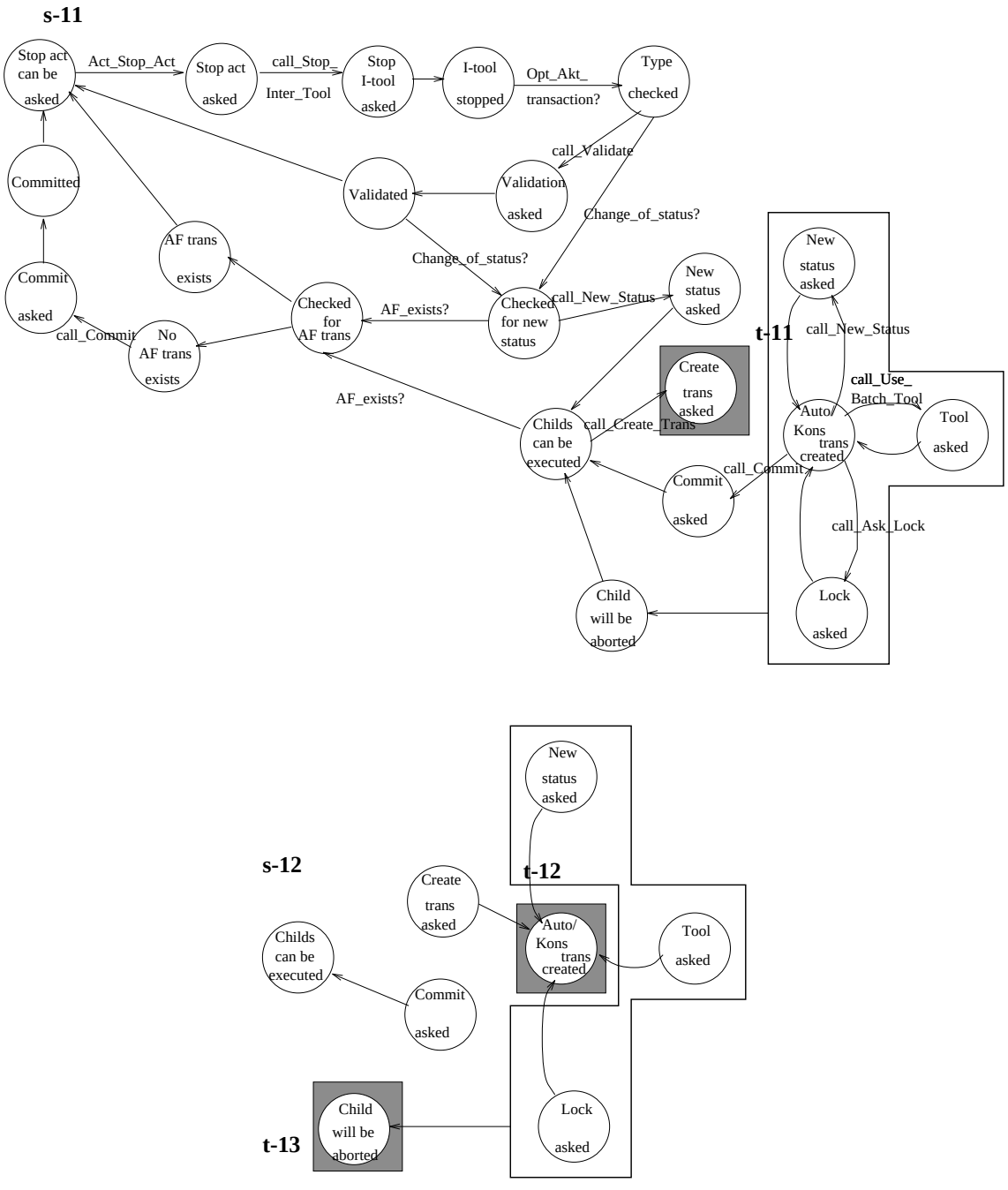
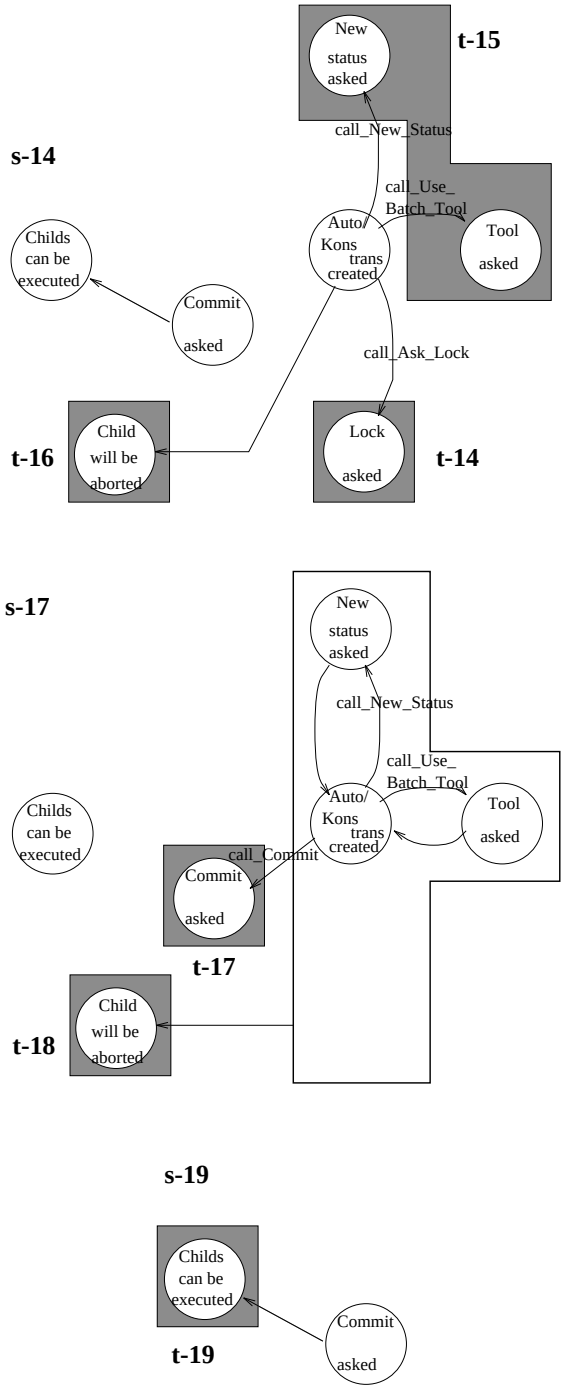


FIGURE 5.5.23. int-Ask_Lock's subprocesses and traps w.r.t. Auto Transaction.

In trap t-8 of subprocess s-8, the internal behaviour can't be activated. If the employee switches to subprocess s-9, the behaviour can be activated and the lock can be obtained or the transaction can be aborted. An Auto transaction can't release locks of other transactions, it can only abort other transactions.




New
status
asked
t-15
call_New_Status
s-14
call_Use_
Batch_Tool
Auto/
Kons
trans
created
Tool
asked
Childs
can be
executed
Commit
asked
call_Ask_Lock
Child
will be
aborted
t-16
Lock
asked
t-14
s-17
New
status
asked
call_New_Status
call_Use_
Batch_Tool
Auto/
Kons
trans
created
Tool
asked
Childs
can be
executed
call_Commit
Commit
asked
t-17
Child
will be
aborted
t-18
s-19
Childs
can be
executed
Commit
asked
t-19

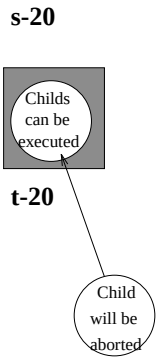
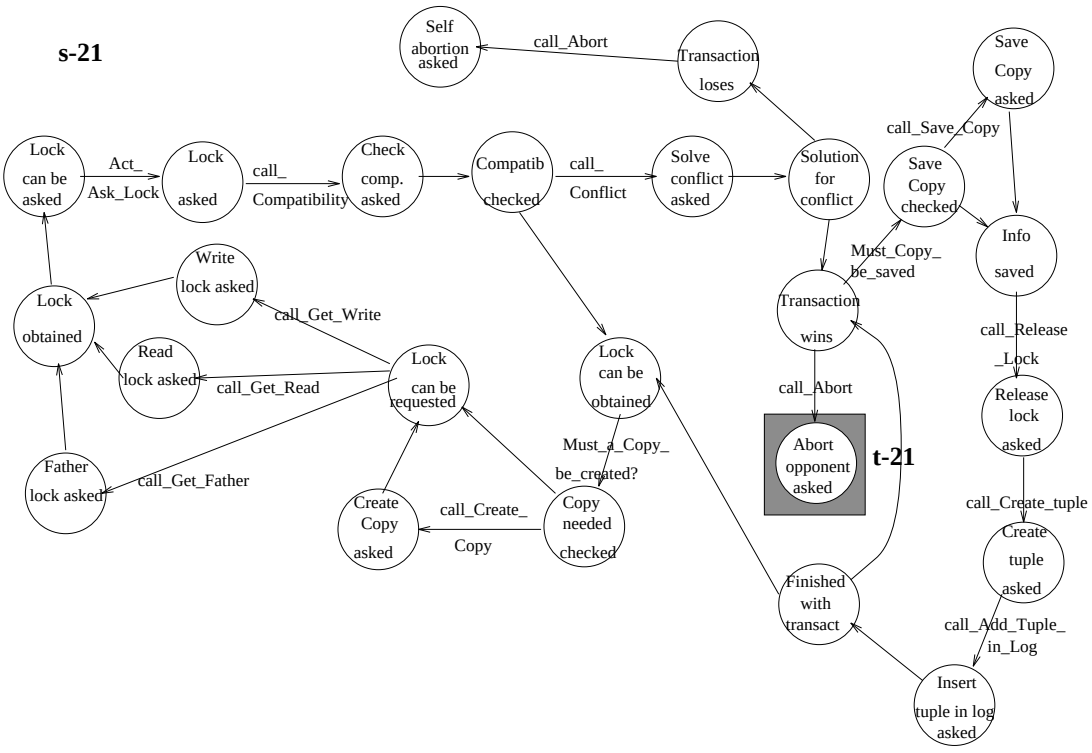


FIGURE 5.5.24. int-Stop_Act's subprocesses and traps w.r.t. Auto Transaction.

In trap t-11 of subprocess s-11, the call is made to create an Auto transaction. Afterwards, the transaction can either start to ask locks (trap t-14) or execute (trap t-15). So, if either New_Status or Use_Batch_tool is called, the behaviour has to be in the state *Execute*. If another transaction aborts this Auto transaction, the trap t-16 will be entered. In trap t-20, the transaction is aborted.



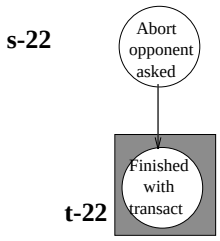


FIGURE 5.5.25. int-Ask_Lock's subprocesses and traps from Kons -, Pess_Akt - or Pess_AF Transaction w.r.t. Auto Transaction.

The above employee has to be an instance of the class Kons -, Pess_Akt or Pess_AF Transaction, because only such a transaction can win a conflict from an Auto Transaction. In trap t-21, a transaction has made the call to abort an Auto transaction.

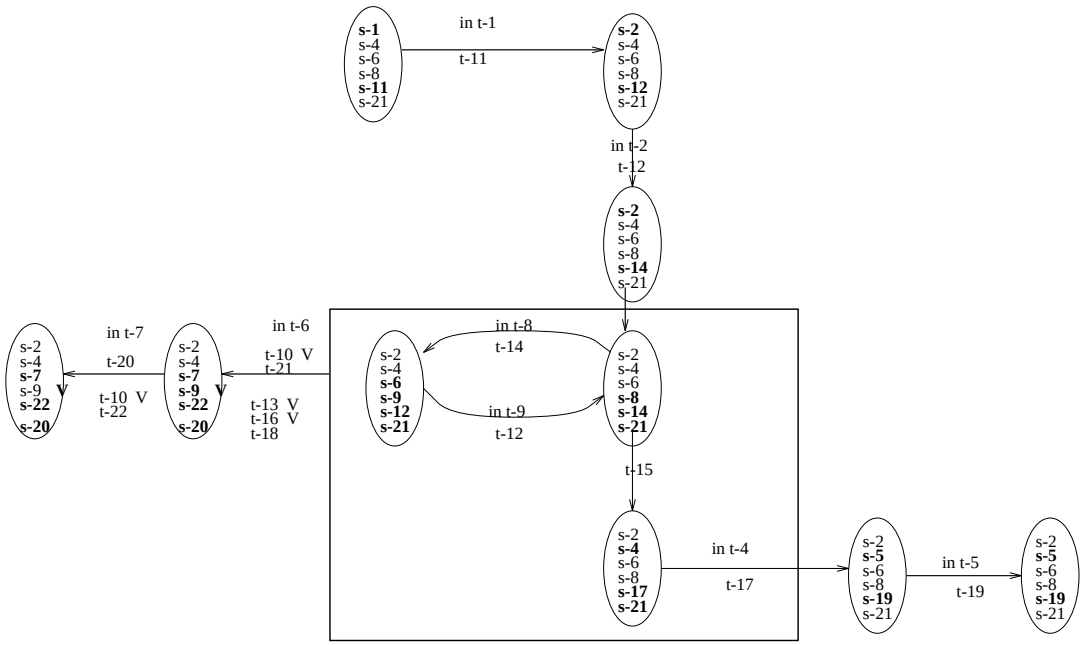


FIGURE 5.5.26. Auto Transaction, manager of six employees.

When the operation Stop_Act is executing and it has created an Auto transaction, another transaction T2 can abort this transaction.
The problem occurs, when Stop_Act is working on an instance of the class Auto Transaction and during its execution another transaction T2 aborts this instance. This means that T2 changes the state of the external behaviour. The employee of Stop_Act must have traps and subprocesses, which don't result from the uses relation. This means that special traps and subprocesses have to be defined to model the influence of the abort, during execution of Stop_Act.
It is not always sufficient to model only the uses relation.

6. Problems and solutions in Socca

During the creation of the previous specification, a couple of weaknesses of the Socca formalism occurred. Some of the weaknesses are small, but these weaknesses are also listed.
First the problems are listed. For those problems, a solution will be proposed. Then, the remarks are given. These remarks point out some special situations extra. Also guidelines, to help the specifier to conquer some classical problems, are listed. If a specifier starts to use Socca, he will often experience some classical problems during the specification process. These problems have to be conquered to rewrite that part of the specification. The guide lines list some classical problems and explain how they can be prevented. At the end, general weaknesses of the Socca formalism are listed.
After each point of comment, the page number with an example in the specification of the described comment is given. It is perhaps not that easy to grasp all the listed points of comment, because they are described in a abstract and general way. Use the page numbers to see an example in the specification of such a point of comment.
After all the weaknesses are listed, a new idea to use PARADIGM in Socca is proposed to conquer some of the problems and weaknesses. In chapter 6.5, this new way to embed PARADIGM in Socca is proposed.

As this thesis is devoted to the identification of weaknesses in Socca, this chapter focuses on those weak sides. The positive aspects are not mentioned here. A more balanced evaluation of Socca is given in the chapter Conclusion.

6.1 Problems

1
It can happen that two or more operations can call the same operation (Op1) of a specific instance, but only one operation actually calls operation Op1 and therefore only one operation starts operation Op1. So, with use of PARADIGM, one operation should call the operation Op1. When the employee Op1 is called and therefore switches from subprocess, one operation, which made the call, also has to switch from subprocess.
A solution to model this it to extend the PARADIGM part. To model, that only one employee exclusively calls Op1 and therefore has to change from subprocess, the exclusive or, a logical operator is used. The exclusive or is represented by the sign V.
(page 67)

2
One has to choose the operations of an external behaviour carefully. Sometimes, it is questionable whether an operation has to become part of a class. It can happen that an instance In1 of class C1 can be affected, because an instance In2 of class C2 is updated. In instance In2, information concerning In1 is written. Must this updating of instance In2 be specified in the external behaviour of C1?
A solution is, that if it is important enough to model this in the external behaviour of class C1, one could specify a dummy operation in the class C1. Each time instance In2 is updated and one wants to model this change in the external behaviour of C1, this new operation should be

called.
(page 78)

3
As already described in 8, it can occur that an operation Op1 of class C1 can be activated from several different states in the external behaviour of C1. Problems occur when several of those operations Op1, which are activated in different states in the external behaviour, stop executing in the same state (let's call this state End_state). The problem is that because such an operation Op1 is called from several different states, at the end of execution for each Op1 in the state End_state some subprocesses can be different.
A solution for this problem is to specify for each Op1, which is activated in a specific state, its own End_state.
Another solution is to specify a new sort of state. In this state, all the subprocesses are the same as the state before the behaviour entered this state. The state End_state will become such a special state. Then, it is no problem whether some subprocesses are different when operation Op1, activated from different states, stops executing in state End_state.
(page 67)

6.2 Remarks

4
During execution of an operation (Op1), another operation (Op2) can influence the internal behaviour of this operation. This is so, because Op1 and Op2 both use the same instance on the same moment. This is a rare situation, that two operations use the same instance on the same moment. The kind of influence on Op1 can be modelled.
(page 83)

5
It can be difficult to model how an operation will be exactly executed. If the operation can follow several ways to its end, it can occur that a couple of complex rules decide which way has to be taken. Incorporating complex rules into the model is difficult, because this formalism is a graphical formalism instead of a textual one. The only way often to explain the way, which is taken, is to discuss this under the diagram in the text.
(page 78)

6.3 Guidelines

6
It is sometimes difficult to specify a trap between states of a manager, because no operation of this class accomplishes the switch from one state to another.
This could be solved, by adding dummy operations. The only task of such a dummy operation is to accomplish the switch.
(page 78)

7
If several subclasses export the same operation Op1, it can occur that when an operation Op2 calls operation Op1, Op2 will always call Op1 in a specific subclass.
The attribute value of an instance of a subclass is important whether it may be called. When operation Op2 never calls operation Op1 in a specific subclass, Op2 will not be used as an employee of the manager of this subclass.
(page 78)

8
A certain class C1 has an operation Op1. It can happen, that this operation Op1 can appear more then one time in the external behaviour of class C1. This means, from more states in the external behaviour, operation Op1 can be activated.
If operation Op2 of another class calls operation Op1, one has to be very careful to specify from which state in the external behaviour operation Op1 is called by Op2.
(page 67)

9
There are two sorts of employees. One sort of employees is of the same class as the manager and one sort of employees is of another class then the manager.
With the second sort of employees, one has to choose the states in the subprocesses carefully. An operation Op1, which calls an operation Op2 of class C2, can use several instances of the class C2 during execution. It would be preferable when at any moment, the operation Op1 is referring to just one instance.
When the employee of operation Op1 switches from subprocess after the call of Op2 is made, it is important to be sure that all the states in this subprocess refer to the same instance of this class C2. It would become more easily a mess, if in this subprocess there is referred to several instances of class C2.
(page 63)

10
It is not allowed that several operations of the same class X in the external behaviour of the class X are activated in all different states and then just after activating enter the same state. If it would be specified, several operations can be activated in this specific state of the external behaviour. But only one operation may be activated, because only one operation is called.

6.4 General weaknesses

11
A manager is difficult to survey, because a lot of information over the different subprocesses is inserted in such a manager.
To structure all the information in such a manager, important subprocesses in a state of a manager are drawn bold. In the less important subprocesses, no execution of the internal behaviour takes place.
(page 67)

12
The PARADIGM part of the specification is too large, therefore it is difficult to survey the total PARADIGM part.

In the next table, it is presented in what manager occurred what problem or what guide line or remark has to be used, to make a good specification.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Time Stamp								
Tuple	×							
Locks	×		×					×
Log	×							
Status	×							
Contents	×							
Kons Transaction		×		×	×	×	×	
Auto Transaction	×	×		×	×	×	×	
Pess_Akt Transaction	×	×			×	×	×	
Pess_AF Transaction	×	×			×	×	×	
Opt_Akt Transaction		×				×	×	
Working Context	×							
Process Engine								

The guide line 9 and the general weaknesses 11 and 12 occur in each manager. The guide line, which is described in 10, doesn't occur in this specification, because when this problem occurred, it was conquered. So, this guide line is used for whole the specification.

Almost all the problems, which are found and listed, arise from the integration of the PARADIGM formalism in Socca. The problems are not typical PARADIGM problems, but integration problems. When we are discussing PARADIGM in this and the next chapters, we mean the PARADIGM embedded in Socca. The embedded PARADIGM is a bit different then standard PARADIGM [Gr 91].
Beside all the problems, which are listed, it is also questionable whether all the communication can be described fully. Using only the uses relation, to describe the communication can be too little. Real answers on this question aren't given in this master's thesis.

Remark 5 isn't that much of a problem. One knows that one loses and wins something, when a graphical specification is used. In my eyes, we have won absolutely more, then we have lost.

6.5 A new version of PARADIGM in Socca

With this new version of PARADIGM, only some weaknesses and problems are discussed. So, this new version is only a step in the right direction.
General weakness 12 is tried to tackle, because in the solution the PARADIGM part will be reduced in size.
General weakness 11 is tried to tackle, because in the solution subprocesses won't be specified separate any more. The employee is specified with one diagram, where in all the traps are given.
Problem 1 is solved, because instead of one view of the manager, two views will be specified. The semantics of annotation at the transition edges in a manager is different for both views.

The first view on the manager of a class will have employees of its own class.
If the behaviour executes a transition in the manager, which is labelled with a trap of an employee, it has to be in this trap.
The second view on the manager will have employees, who call employees of this class.
If the behaviour executes a transition in the manager, which is labelled with a trap of an employee, it can be in this trap. All the traps or no traps of some employee will be completely obeyed. Which employees will be obeyed depend on which operations use this instance of the class. In this way, problem 1 is solved.

The solution will be presented and enlightened with the PARADIGM specification of the class Auto Transaction.
The first view on the manager of class Auto Transaction has the following employees:

Create_Trans
Commit
Abort
Ask_Lock

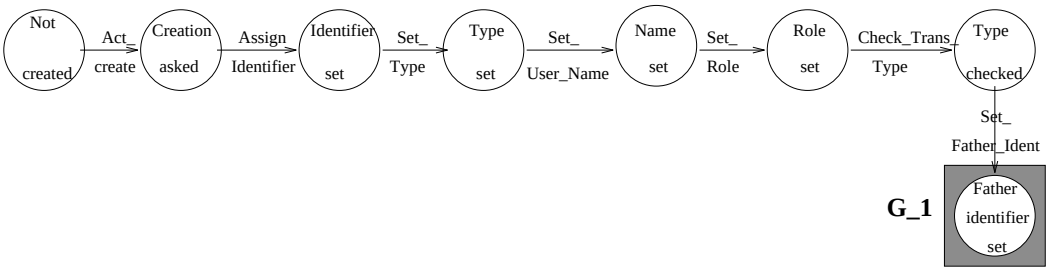


FIGURE 1. int-Create_Trans's goals w.r.t. Auto Transaction.

If one studies the PARADIGM part of the specification or the appendix of this thesis, one can find a pattern in the way subprocesses and traps are specified for an employee of the same class as the manager.
The pattern is, that in the first subprocess, it is waiting to activate. The manager allows the employee to switch from subprocess and the employee will execute and reach its goal, which is a trap.
In the above diagram, the employee is first in state *Not created*. If it is started by the manager,

it is allowed to activate the operation. If it has reached its goal, the employee is in G_1.
In G_1 is the Auto transaction created.
Goal G-1 can be compared with a trap in standard PARADIGM. As one can see, the trap for the state *Not created* is not specifically given in the above diagram. If the manager wants to activate the operation, the internal behaviour of the operation can be in only one state. In the above case is this state *Not created*.

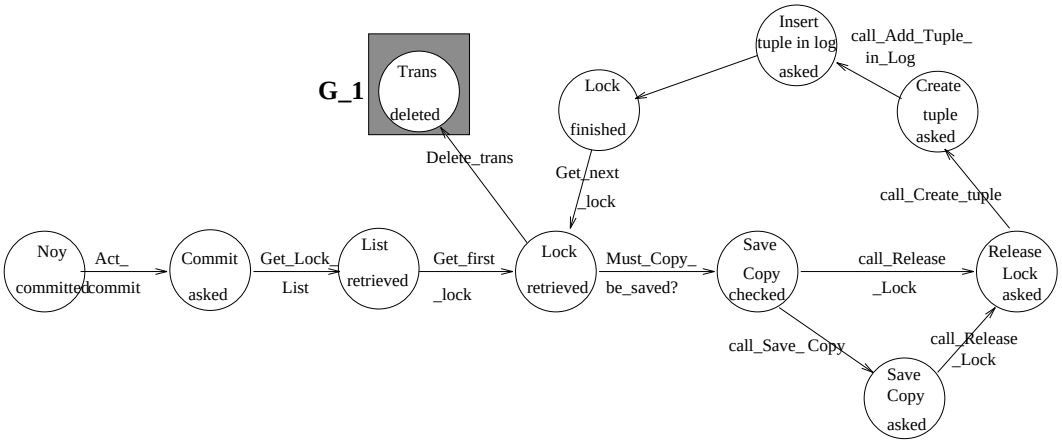


FIGURE 2. int-Commit's goals w.r.t. Auto Transaction.

In G_1, the employee has reached its goal and therefore the transaction is committed.

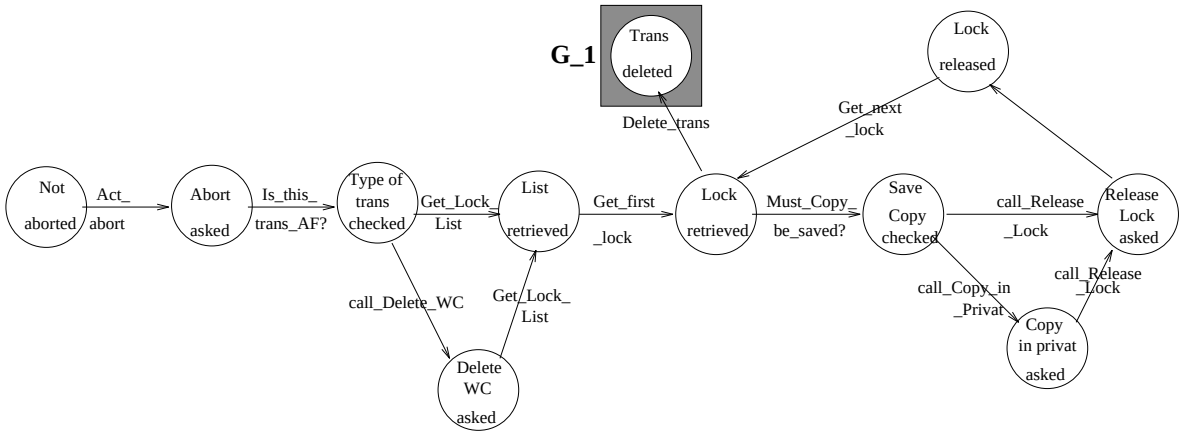


FIGURE 3. int-Abort's goals w.r.t. Auto Transaction.

In G_1 is the transaction aborted.

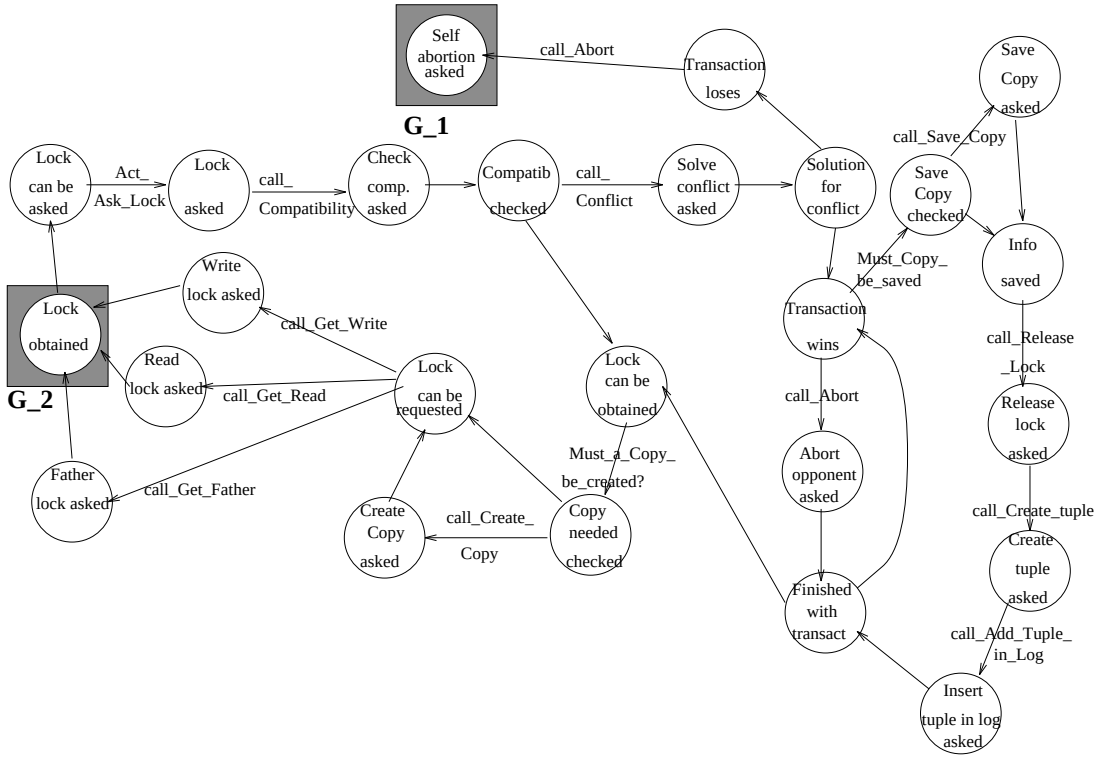


FIGURE 4. int-Ask_Lock's goals w.r.t. Auto Transaction.

This operation has two goals. It either aborts itself, because it couldn't obtain a lock or it has obtained a lock. In G_1 will the transaction abort itself. In G_2 is the lock obtained.

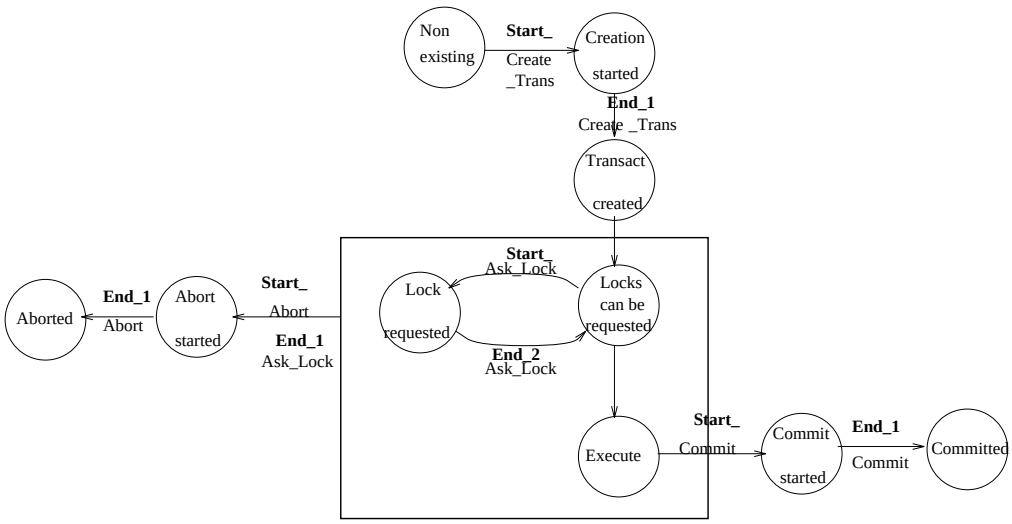


FIGURE 5. Auto Transaction, first view on manager with four employees.

If a transition is labelled with Start and then an operation name. It means that the operation may be activated. In the standard PARADIGM part, the employee now would switch from subprocess. The operation can now be executed. When the transition End, with a number and then the operation name, is reached, it means that the operation must be finished with executing and has reached the goal with this specific number. As already explained, an employee has

numbered its goals.
Reaching a goal can be compared, with reaching the trap in the (often) second subprocess. The labels of the manager of figure 5 can be compared with the traps in standard PARADIGM. The subprocesses in the states of the manager are disappeared. Now, in the states are the names of the external behaviour. This will make it more clear, why what operation is started or ended.

All the labels (traps) of a transition in the manager have to be obeyed, when the manager uses this transition. In the left side of the figure for instance, both Ask_lock must reach goal G_1 and Abort must be activated.

The second view on the manager of the class Auto Transaction has two employees. These two operations call operations of class Auto transaction.

Stop_Act
Ask_Lock

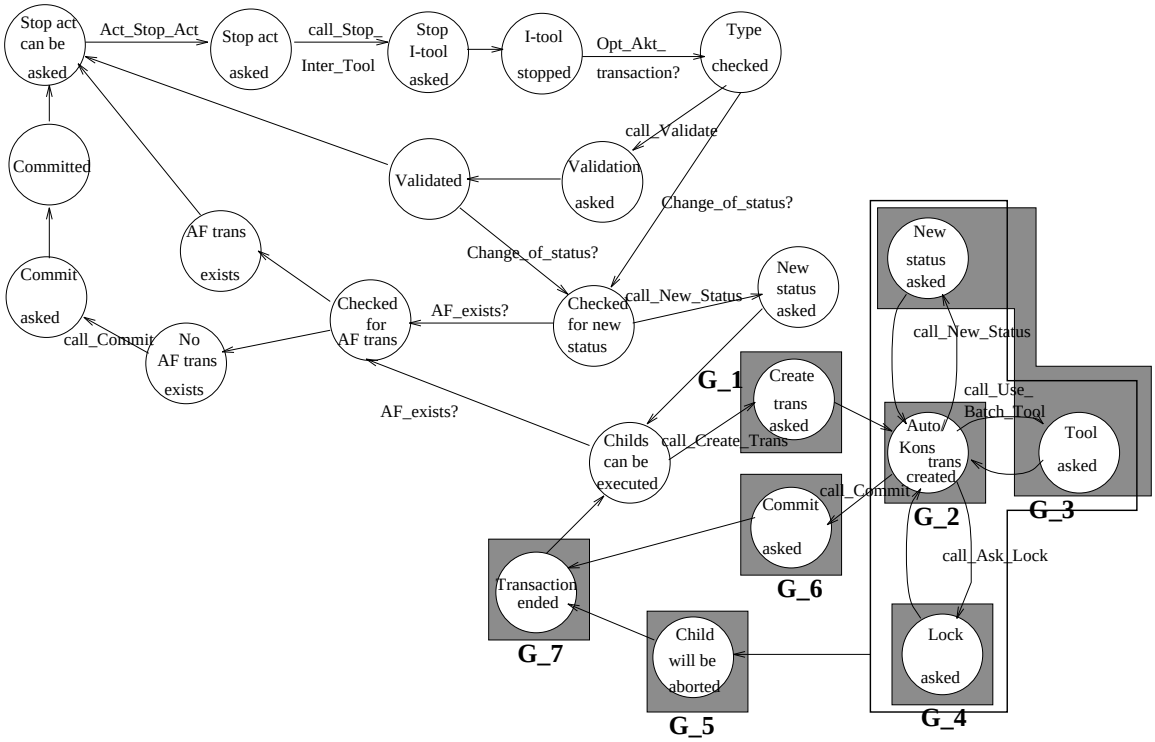


FIGURE 6. int-Stop_Act's goals w.r.t. Auto Transaction.

An operation, which calls an operation of class Auto Transaction, has several goals. After one goal is fulfilled, often a new goal has to be reached. In the above employee are seven goals defined. This employee can be compared with the employee, which is defined in standard PARADIGM. Each goal is in standard PARADIGM a trap. Just like how it is with a trap, one can leave the goal, when the manager has said it must have reached this goal. Now, the next goal can be reached.

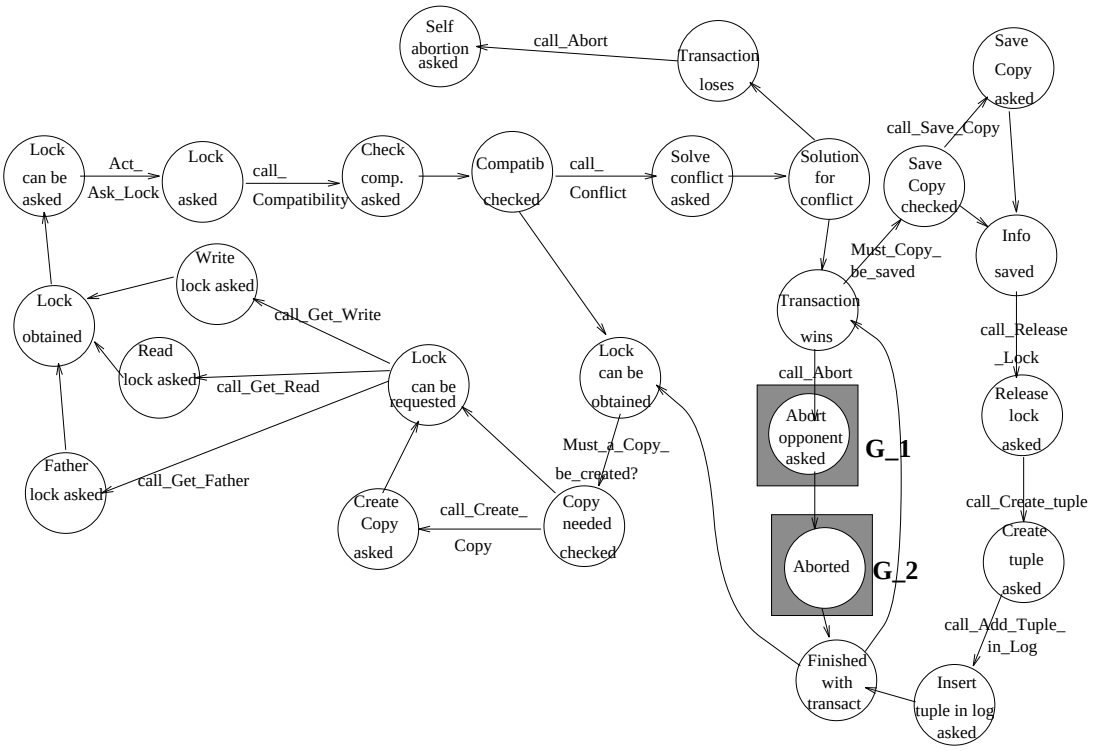


FIGURE 7. int-Ask_Lock's goals w.r.t. Auto Transaction.

In goal G_1, the call is made, to abort the transaction. In goal G_2, the transaction is aborted.

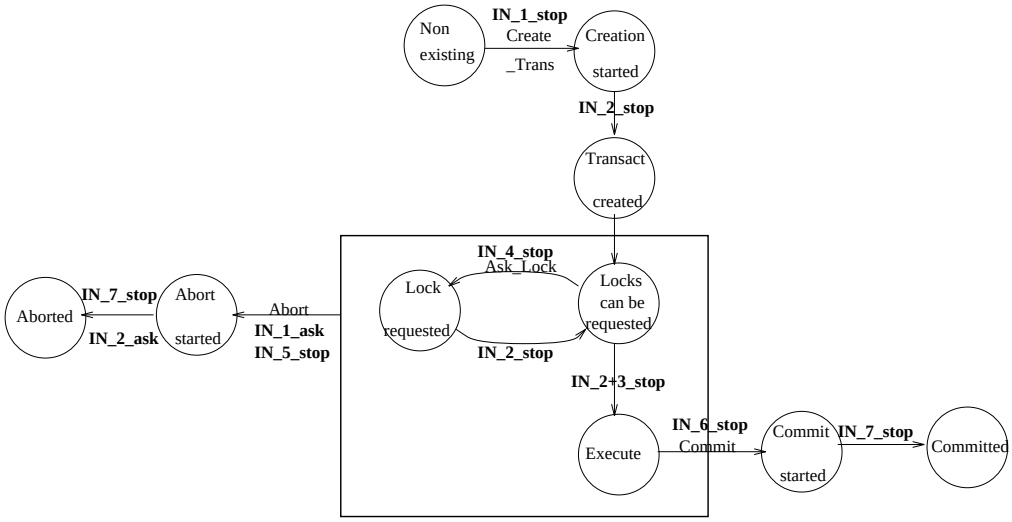


FIGURE 8. Auto Transaction, second view on manager with two employees.

Each transition has a label, what prescribes in what goal the employee has to be. Each label consists of the word IN, then the goal and then the name of the operation. There is one special goal, this is IN_2+3_stop. This goal is build up of two goals. The behaviour has to be in this constructed goal.
Furthermore, the manager doesn't have to fulfil all the goals. In the transition between the states *Abort started* and *Aborted*, it is possible that either one or both goals have to be fulfilled.

If the Auto transaction is aborted by another transaction, both goals have to be fulfilled. If the transaction aborts itself, only the goal IN_7_stop has to be fulfilled. Whether goals have to be fulfilled or not depends on whether the operation, which is related to such a goal, calls an operation of class Auto Transaction.

With this new approach to use PARADIGM in Socca, some information is lost compared to the old version. The older version specified the coordination on a very fine level. There is no reason why the level of abstraction can not be lifted a little bit.
For the above example, the new version works very good. One example can however be given, that a problem occurs with this new version. This is the case, when between two goals two or more paths in the internal behaviour can be used. If the employee wants to reach the new goal, it is not specified what path has to be taken. In the old PARADIGM version, it was made sure that when paths are illegal, some states of the subprocesses were omitted. In this new PARADIGM version, one could specify also what path can be taken between two goals in an employee, if one has two or more paths between the goals and it has to be specified what path can be taken by the employee. For each employee, paths are specified if it is necessary. At the transition edge of the manager, it is specified what path or paths may be taken to reach the new goal.

With this new method, a big reduction in size of the PARADIGM part can be reached. In the above case, a reduction of 50% is achieved. Normally, the reduction will be a bit smaller. Besides the reduction, the managers are better understandable and problem 1 (the exclusive or problem) is solved, because two views for one manager are used.
In the above solution, the ideas behind PARADIGM aren’t really changed. The solution manly proposes another way to visualize PARADIGM.

Besides weaknesses and comments, Socca has also many positive characteristics. The strength and my personal view on Socca will be taken into account in the chapter Conclusion. In the next chapter, the weaknesses of how the dissertation is structured, compared to the way the same theory is specified in Socca, will be identified.

7. Motivation for using Socca

Besides testing Socca, another goal is defined in the beginning of the thesis. This goal is what the strengths are to use a specification language, in this case Socca, to describe a concept. The concepts, used for this specification, mainly can be found back in [Wo 94]. In this dissertation, one part was textual description and the other part was code of the process transaction model.

We want to keep this chapter short. Not because, it isn't an interesting subject to talk about the usefulness of a specification language. In our eyes, this is a very interesting subject. The reason is that already many important things are said and saying more can only distract the attention to less important things. Still, some very important results have to be given how the Merlin theory could be described with Socca.

My personal experiences during specification of the process transaction model with such a way of describing a concept, are given bellow. These points are difficulties, which we had with using a textual description and code to describe the Merlin concepts in the dissertation and how Socca provides more comfort for the reader to understand the theory.

To find information back in code is very difficult. If one wants to learn how the process transaction model fits together, one needs a problem description on a higher level, then the Prolog-like language. A specification will enable the reader to abstract information easier, then the Prolog-like rules.
A textual description of the concepts will (normally) enable the reader to understand the problem pretty fast and the reader won't be bothered with details. But the textual description is often not complete. Sometimes, the description is in contradiction with itself, or some parts of the concept is missing in the description. A specification is more complete then the textual description and by reading the data perspective and external behaviour of Socca, the reader also understands the main parts of the problem.
One of the main characteristics of this specification is the separation of concern, what makes the specification well structured. Therefore, it is easy to find information in the specification. The specification has a top down structure. If one wants to know a detail, one will normally find that in the internal behaviour. Important (static) information can be found faster in the data perspective. The separation in classes makes the search for information even easier. It is now more understandable where and who initiates an operation. Besides who initiates an operation, the coordination between all the operations is also modelled in the PARADIGM part.

In general, the concepts in the dissertation of Stefan Wolf [Wo 94] is consistent and not in contradiction with itself. There are nevertheless some small points of comments over the usefulness to model some parts like they are in the process transaction model.

Synchronization rule 1 is partly in contradiction with the rest of the theory.
In this rule, it is prescribed that write access on a contents of a document by a transaction of the type Opt_Akt, Pess_Akt or Pess_AF will have as result that a lock or time stamp on the status of this document has to be set.
The word write has to be left out, because afterwards it appeared to be the case, that a transaction of the type Opt_Akt, Pess_Akt and Pess_AF always has access on both the contents and status of the document.

Until now, no private directories are implemented in the Merlin prototype. But nevertheless, the contents, worked on by an aborted transaction, has to be saved in a private directory.

As already mentioned, the identifier of a transaction has to be inserted in the log according to the theory in [Wo 94]. In the tuple in this specification, it is left out that the identifier is one of the information items in the tuple. It is unnecessary information to put this in the log.

In the last two chapters the achieved Merlin and Socca goals are evaluated. In general, only the weak sides are discussed. In the next chapter Conclusion, concluding remarks over Socca and Merlin will be given.

8. Conclusions

If one wants to construct software, it would be wise to follow certain steps. First, one has to ask the clients for the problem description. Then, the specifier makes a specification and the client will explain his wishes further during the construction of this specification. Finally, the software will be constructed based on this specification.

A problem with a lot of specification languages is, that somebody without a computer science background has difficulties to understand the specification. This somebody is often the client, who orders a piece of software. Because of this, the specifier and the client have difficulties to understand each other. The client is not sure, that the specifier understands what he means, because the client doesn't understand the specification (very well).
Those specification languages focus to much on the transformation of specification to software. The result can be a good piece of software, but not that what the client wants.
Lotos and VDM are for instance textual specification languages, which are code orientated and not really client friendly.

Socca, on the other hand, is better understandable for people without a computer science background. Using diagrams is a good step in the direction of client friendly specifications. Furthermore, EER and STD formalisms in Socca are very classical and because of that well known. Data perspective and perhaps the external behaviour can be understood by such a client. In such a way, the client has a better overview what will be constructed and the client can give feedback. Of course is the last part of Socca not that easy to understand for somebody without a computer science background. But the later part is already very much in detail.
We believe that the transformation of Socca into code is also not that difficult. All the important operations in the software are specified in the internal behaviour and PARADIGM describes the communication between all those operations.
In my eyes Socca focuses in the right amount on the development of software by the programmer and the problem description with the client.

We really believe in the main ideas behind Socca, but this doesn't mean that whole the Socca formalism works smooth. In chapter 6, we have talked about the weak sides of Socca. The main conclusion of chapter 6 is, that there are some integration problems between PARADIGM and the rest of Socca.
A solution is proposed to try to integrate PARADIGM better in Socca. When I was developing the PARADIGM part in Socca, I noticed that it took to many paper to explain what had to happen. My idea was to look whether the subprocesses perhaps could be left out. If no subprocesses would exist any more, I could also tackle another problem. I wanted to make the managers more easy to survey. The result of leaving subprocesses out of the PARADIGM part in Socca can be read in chapter 6.5.
Still, one problem existed. Could those disgusting exclusive or's in the manger disappear? I noticed, that the differences between employees of the class of the manager and employees of another class then that one of the manager is bigger then expected. These two sorts of employees have to be divided. I divided them in two views of the same manager. For each view, other rules exist how the employees have to behave them self.
With these new ideas, PARADIGM can be integrated better in Socca. Only in the size of paper

to describe PARADIGM in Socca, a reduction of 40% or more can be achieved. Besides reduction in paper, it is now easier to read and understand the manager and the employees in the PARADIGM part.

Furthermore, in chapter 6.3 are guide lines listed. The goal of these guide lines is to make sure, that the specifier doesn't make some classical mistakes.
Besides a solution for the integration problem in chapter 6.5, a new (State chart like) feature is introduced in this thesis. In some STD's of the external behaviour, polygons are used to make figures less crowded. With use of this polygon, several arrows can be left out of the diagram. This is discussed in chapter 4.

If one compares the specification of the process transaction model with the dissertation [Wo 94], one compares mainly how a piece of theory can be described. One can see quit easily the difference between the specification language Socca and the textual description and Prolog-like rules in the dissertation. The Prolog-like rules are a description on a low level and therefore difficult to understand and not client friendly. The textual description is on this moment perhaps the most common used specification language. In a textual description, there is a lack of structure. Because of this, the described theory is in contradiction with itself or some parts of the theory are missing or are difficult to find in the description. As already motivated in chapter 7, using Socca to describe the concepts of the dissertation is more preferable then Prolog-rules and plain text.

It is not that difficult to conclude, that it is preferred to use a real specification language, when one wants to describe a piece of theory. As already pointed out, Socca is a very interesting alternative to use as a specification language.

9. Literature.

[EG 93] G. Engels, L.P.J. Groenewegen, *Specification of Coordinated Behaviour in the Software Development Process*, Proceedings of the 2nd European Workshop on Software Process Technology (EWSPTZ), Trondheim, Norwegen, Lecture Notes in Computer Science Bd. 635, Springer, Berlin, 1992.

[Gr 91] L.P.J. Groenewegen, *Parallel Phenomena 1 - 14*, University of Leiden, Dep. of Computer Science, Tech. Rep. 86-20, 87-01, 87-05, 87-06, 87-11, 87-18, 87-21, 87-29, 87-32, 88-15, 88-17, 88-18, 90-18, 91-19. 1986-1991

[JPSW 94] G. Junkermann, B. Peuschel. W. Schäfer, S. Wolf, *Merlin: Supporting Cooperation in Software Development through a Knowledge-based Environment*, A. Finkelstein, J. Kramer, B. Nuseibeth (Hrsg.), *Software Process Modeling and Technology*, Research Studies Press, John Wiley and sons, England, 1994.

[PS 92] B. Peuschel, W. Schäfer, *A Knowledge-Based Software Development Environment*, International Journal of Software Engineering and Knowledge Engineering, Vol. 2, No.1, pp 79-106, March 1992.

[PSW 92] B. Peuschel, W. Schäfer, S. Wolf, *A knowledge-based Software Development Environment Supporting Cooperative Work*, International Journal of Software Engineering and Knowledge Engineering (SeKe), Vol.2, Nr. 1, World Scientific Publishing Company, March 1992

[Wo 94] S. Wolf, *A transaction-based approach to support cooperative software development*, PhD.-Thesis (in German), University of Dortmund, Department of Computer Science, Software-Technology, 1994

10. Appendix

In this appendix, the whole PARADIGM part is given. For all the managers and their employees, the diagrams are presented, without any textual explanation. Down below, all the classes are listed and behind each class name stands a page number. This is the number where one can find the first employee of this class. For the class Process Transaction, no manager exists. But the three operations of this class are employees for all its five subclasses.

The first manager will be TimeStamp. The employees of this manager are:

Create_TS
Clean_TS
Set_TS
Validate

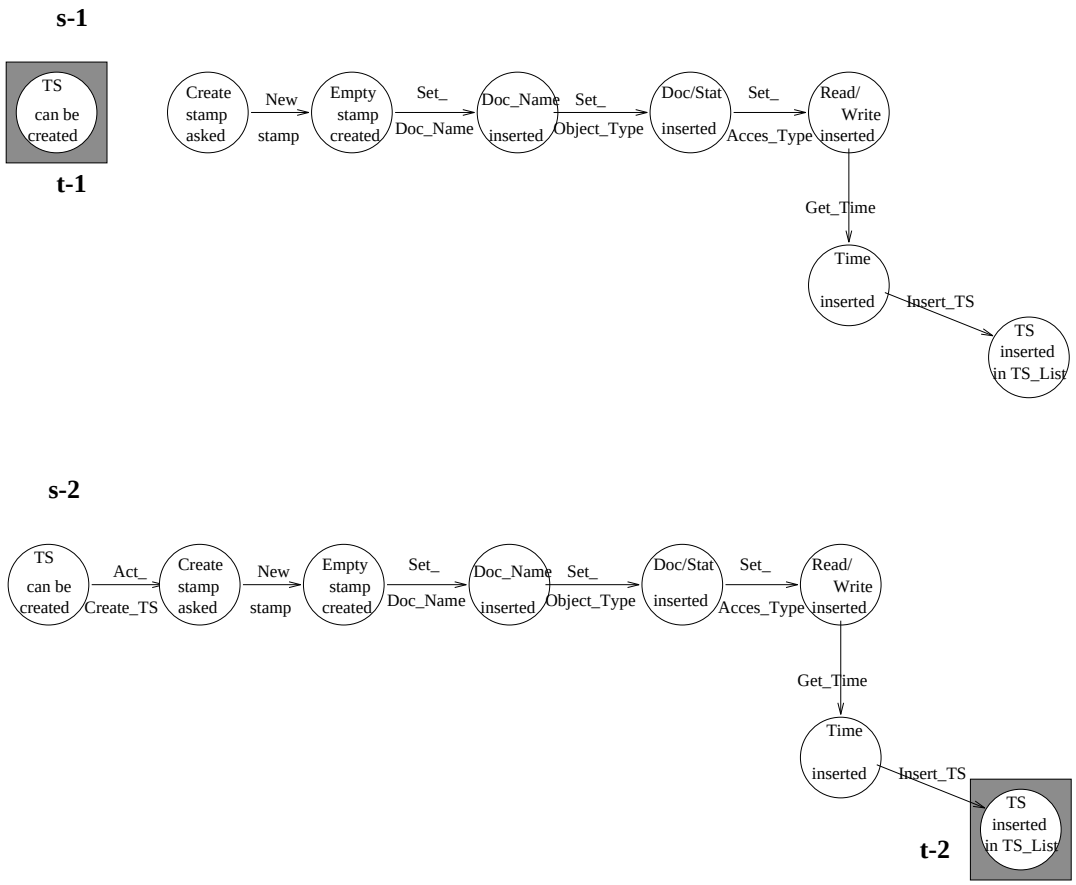


FIGURE 1.int-Create_TS's subprocesses and traps w.r.t. TimeStamp.

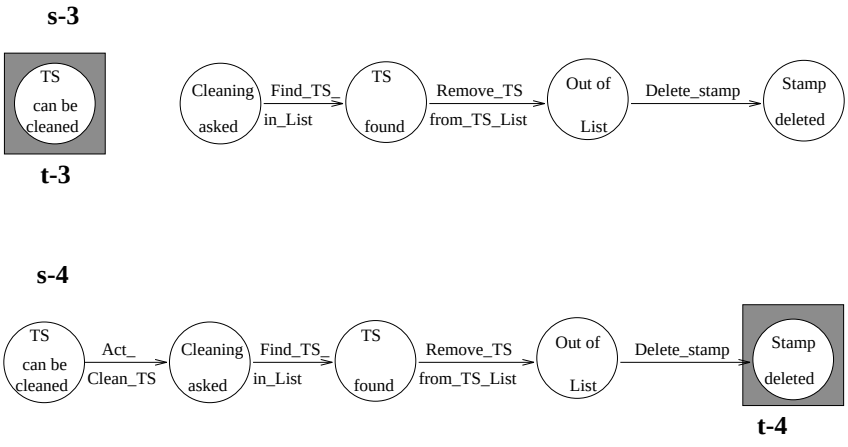


FIGURE 2.int-Clean_TS's subprocesses and traps w.r.t. TimeStamp.

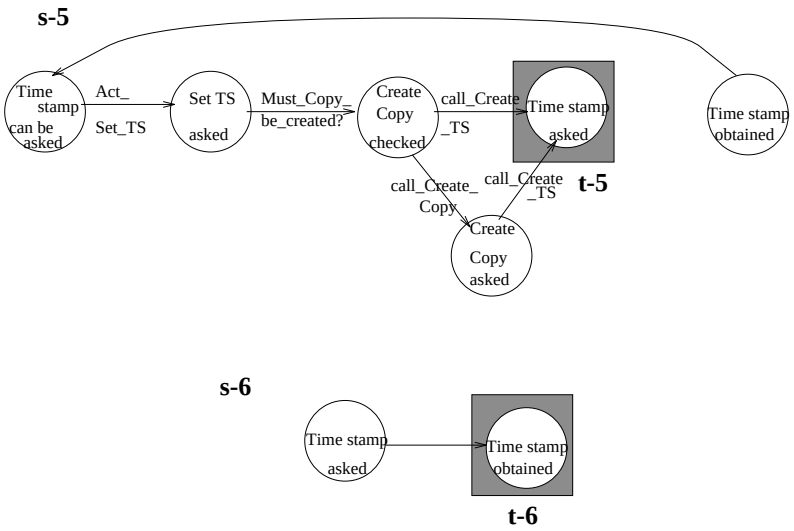
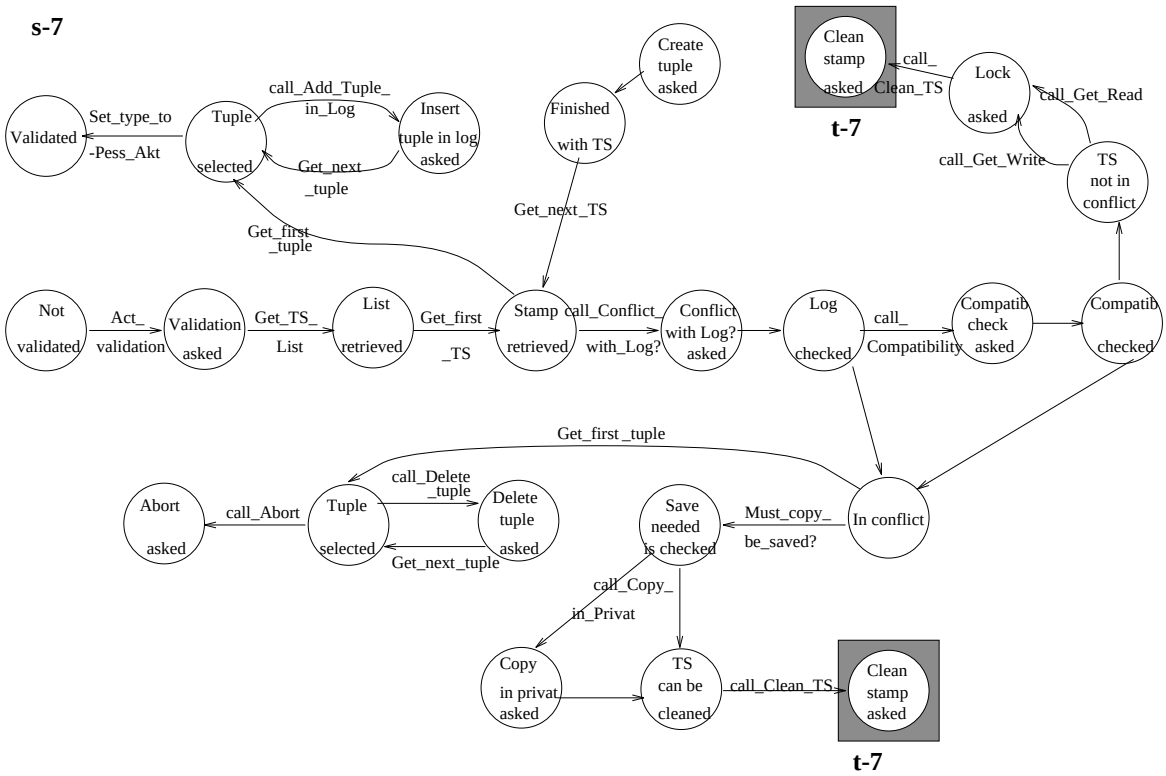


FIGURE 3.int-Set_TS's subprocesses and traps w.r.t. TimeStamp.



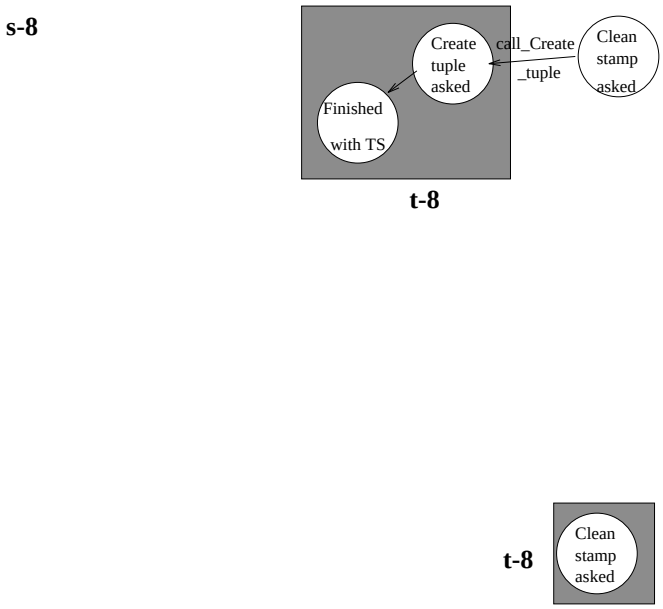


FIGURE 4.int-Validate's subprocesses and traps w.r.t. TimeStamp.

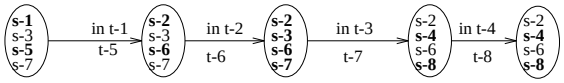


FIGURE 5. TimeStamp, manager of four employees.

The next manager is Tuple. The employees of this manager are:

Create_tuple
Insert_tuple
Delete_tuple
Conflict_with_Tuple?
Add_Tuple_in_Log
Conflict_with_Log?
Commit
Validate
Remove_Lock
Update_Lock

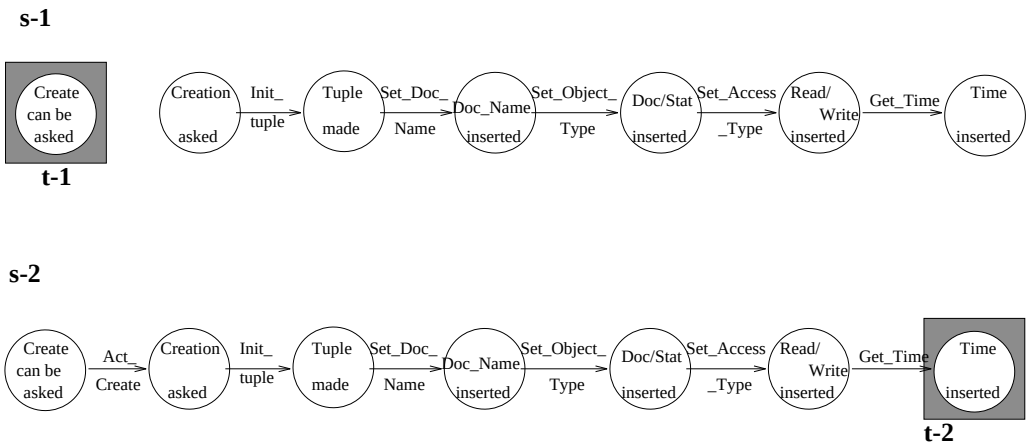


FIGURE 6.int-Create_tuple's subprocesses and traps w.r.t. Tuple.

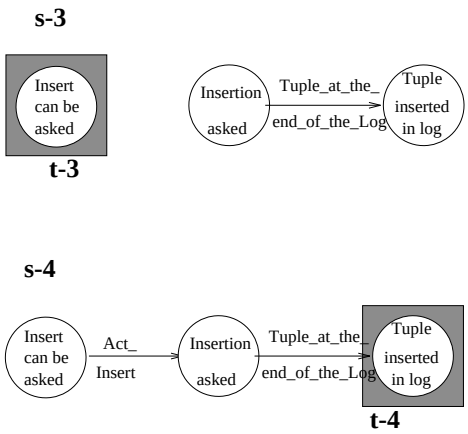


FIGURE 7.int-Insert_tuple's subprocesses and traps w.r.t. Tuple.

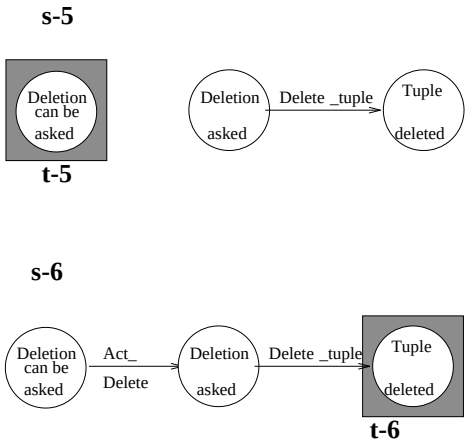


FIGURE 8.int-Delete_tuple's subprocesses and traps w.r.t. Tuple.

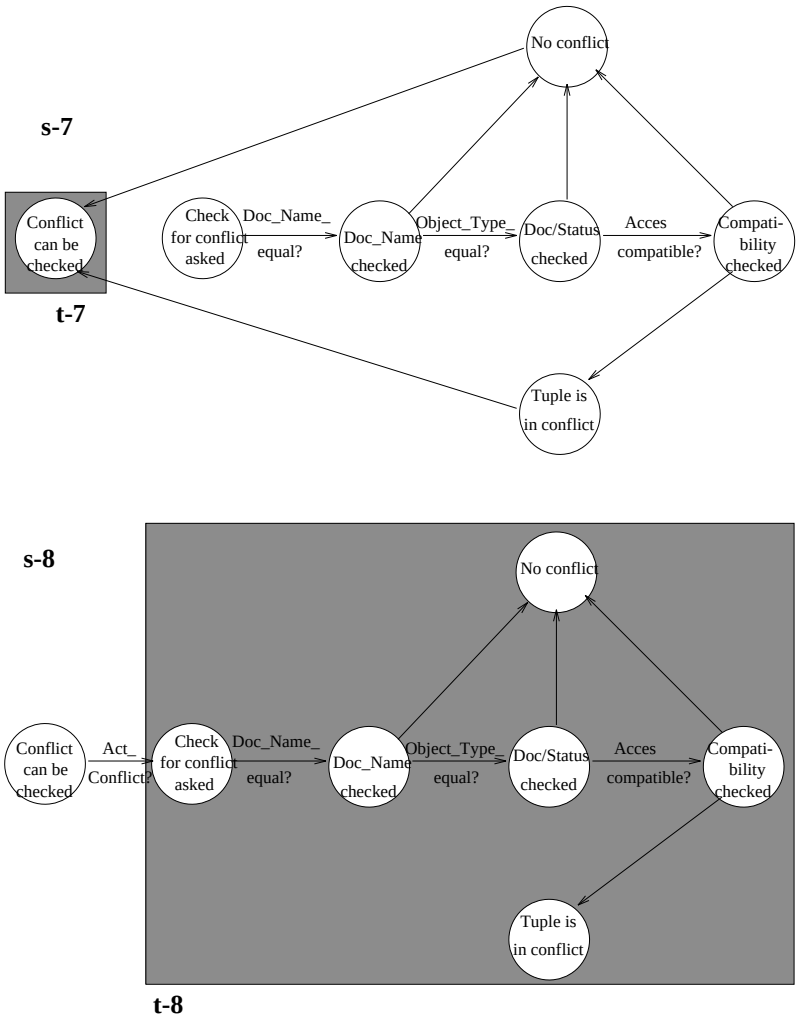


FIGURE 9.int-Conflict_with_Tuple's subprocesses and traps w.r.t. Tuple.

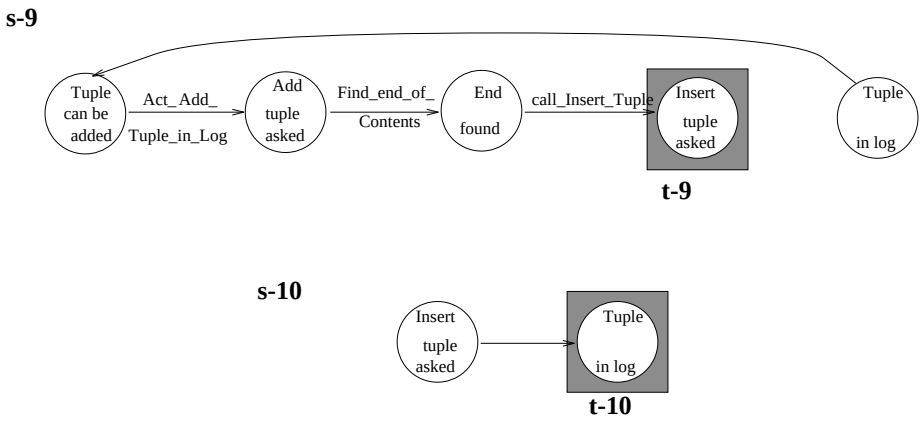


FIGURE 10.int-Add_Tuple_in_Log's subprocesses and traps w.r.t. Tuple.

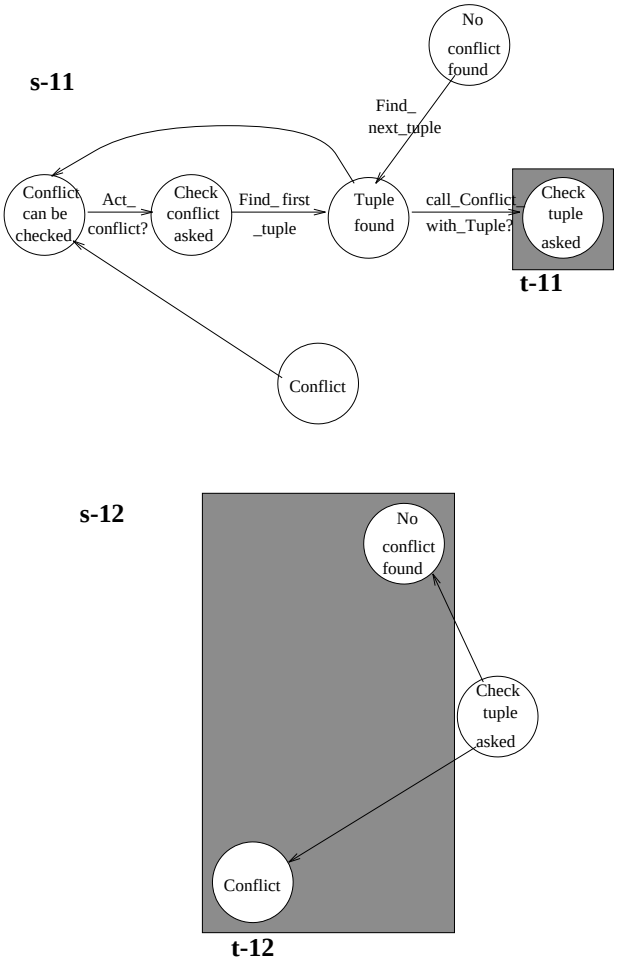


FIGURE 11.int-Conflict_with_Log's subprocesses and traps w.r.t. Tuple.

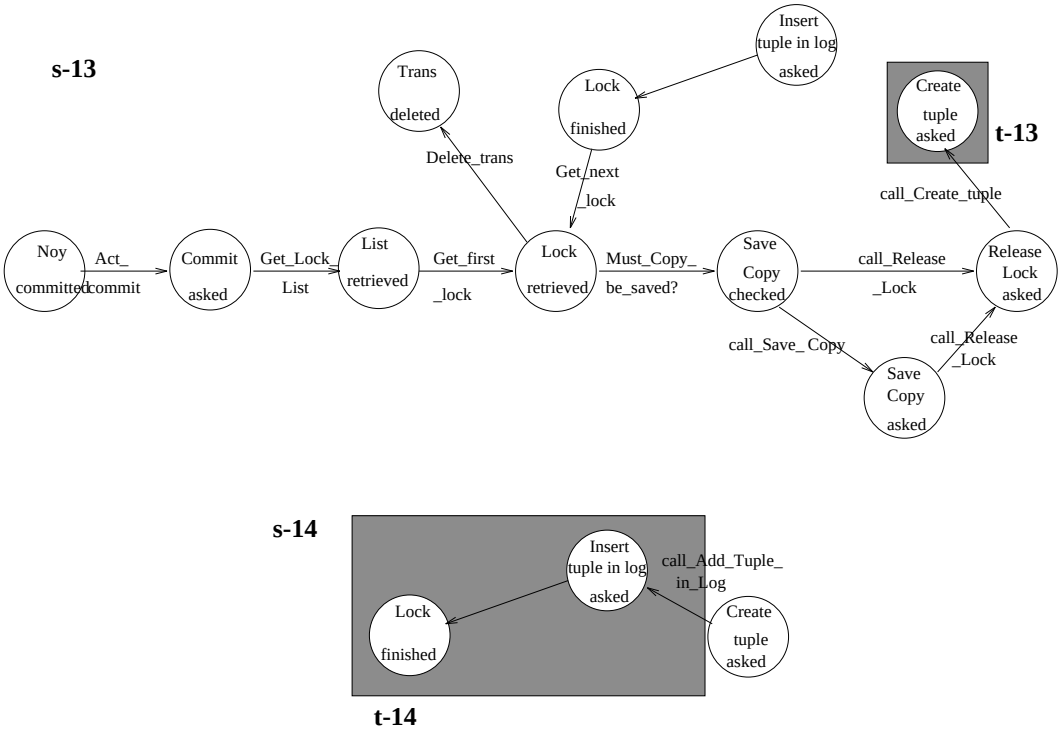
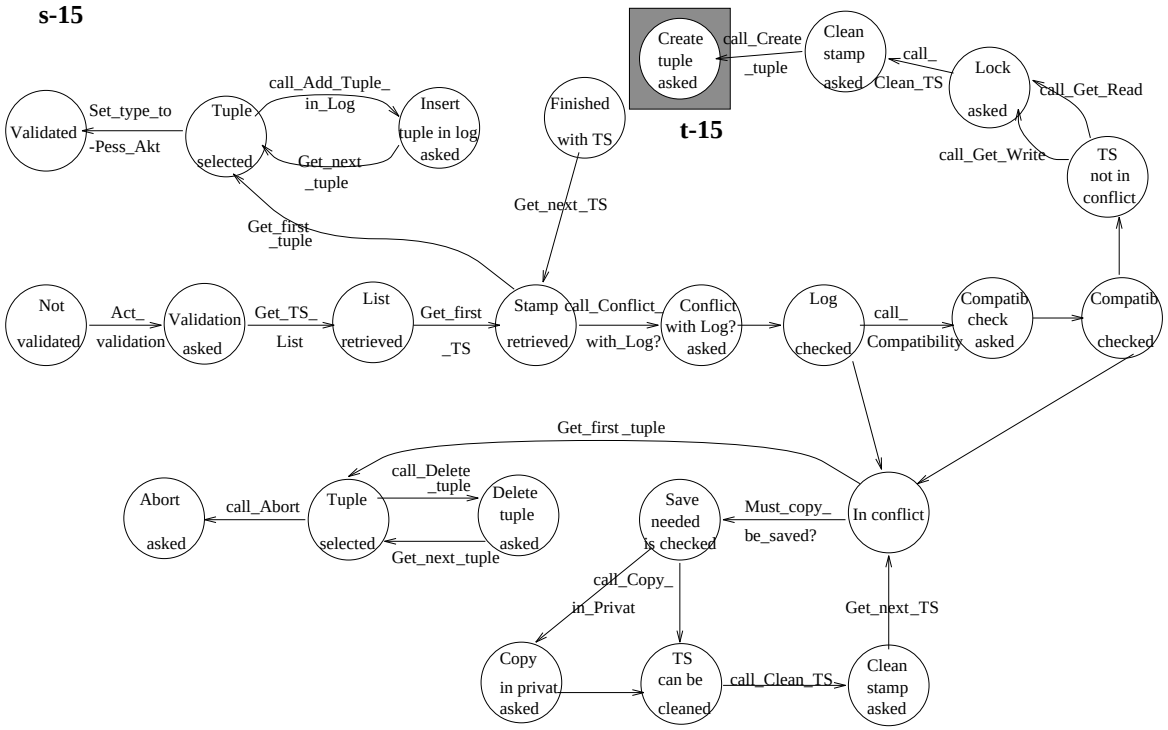


FIGURE 12.int-Commit's subprocesses and traps w.r.t. Tuple.



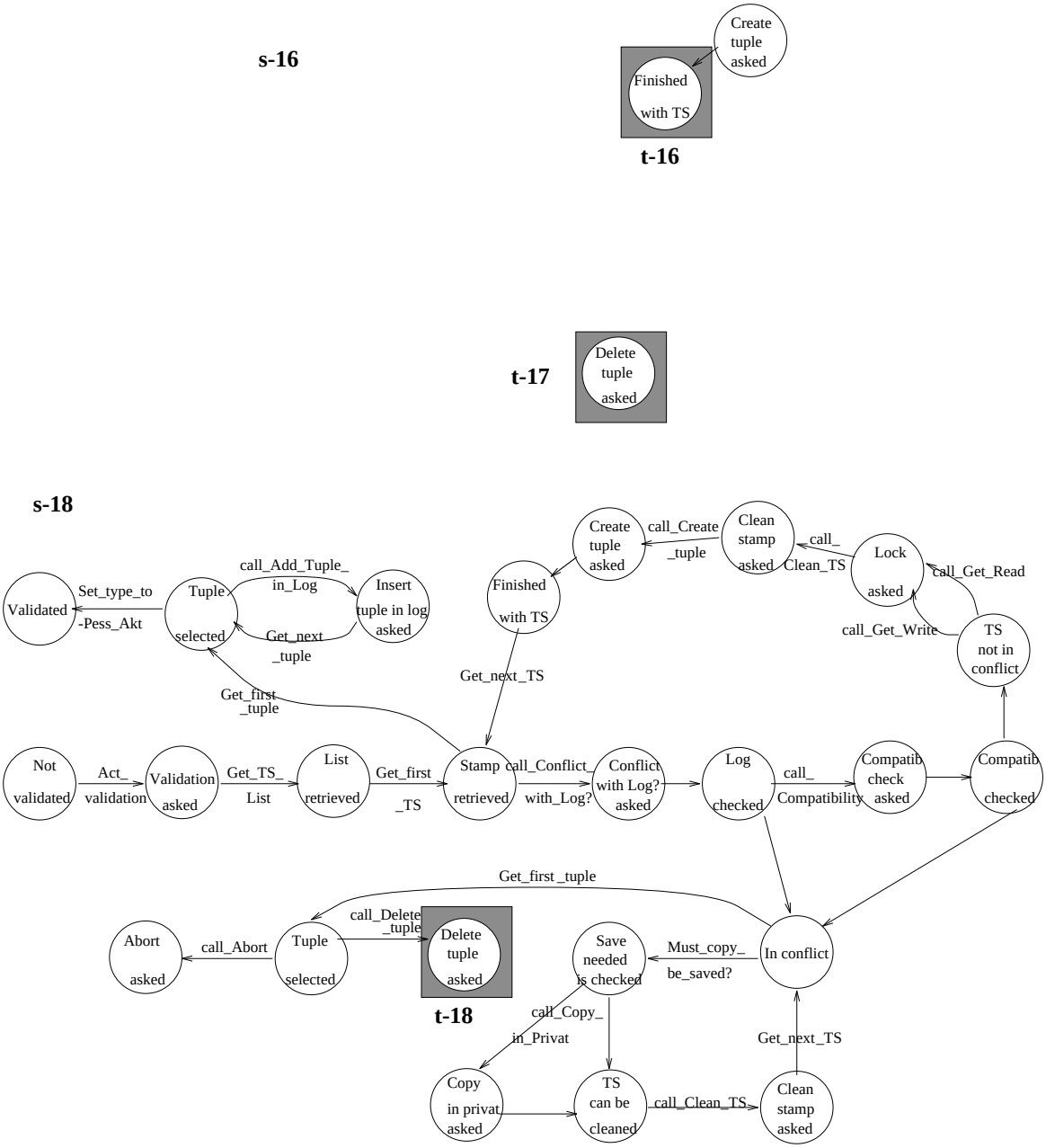
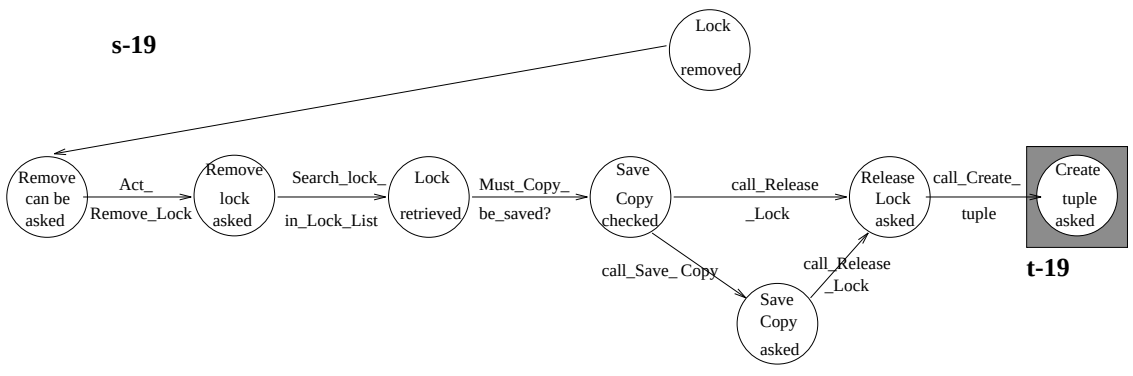


FIGURE 13.int-Validate's subprocesses and traps w.r.t. Tuple.



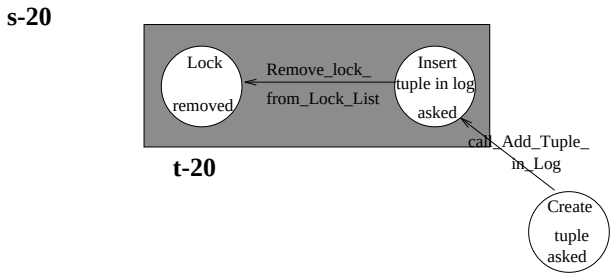


FIGURE 14.int-Remove_Lock's subprocesses and traps w.r.t. Tuple.

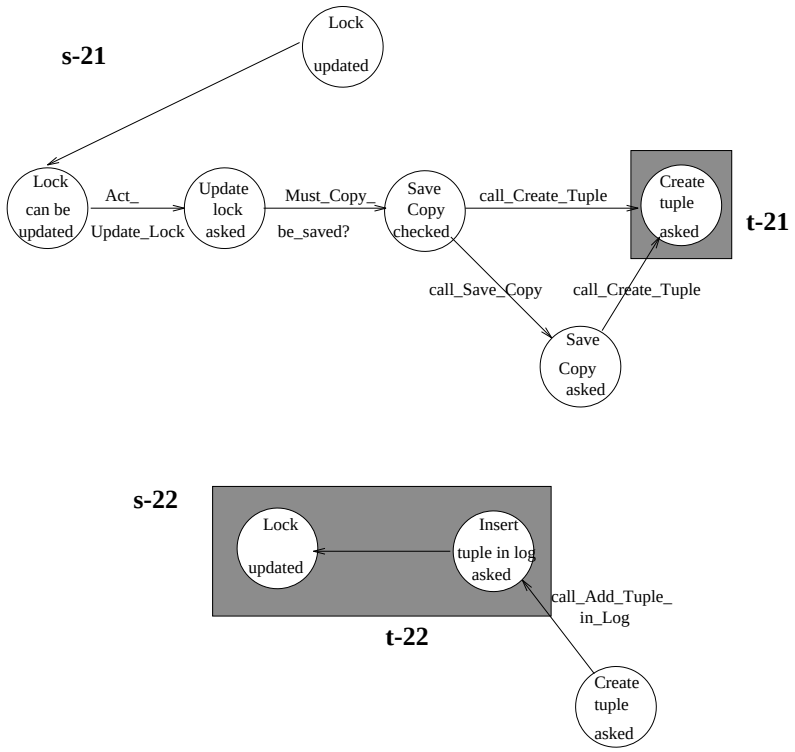


FIGURE 15.int-Update_Lock's subprocesses and traps w.r.t. Tuple.

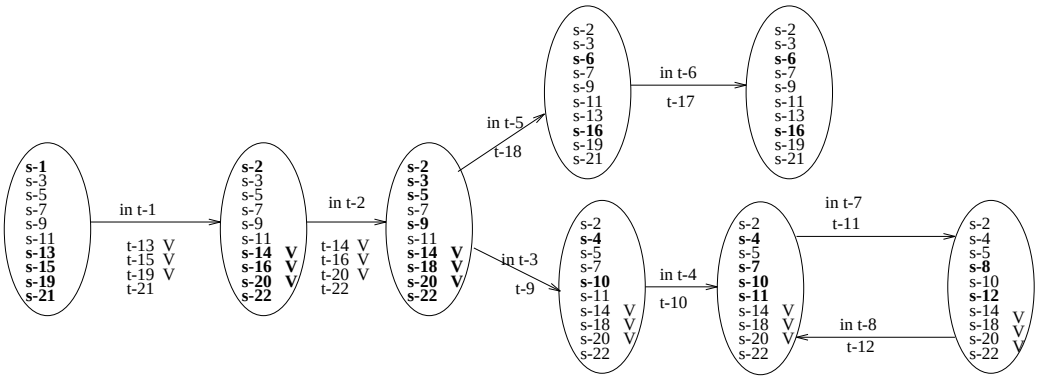


FIGURE 16. Tuple, manager of ten employees.

The next manager is Locks. The employees of this manager are:

Get_Read_Lock
Get_Write_Lock
Get_Lock_Father
Release_Lock
Compatibility
Conflict
Ask_Lock
Commit
Remove_Lock
Abort

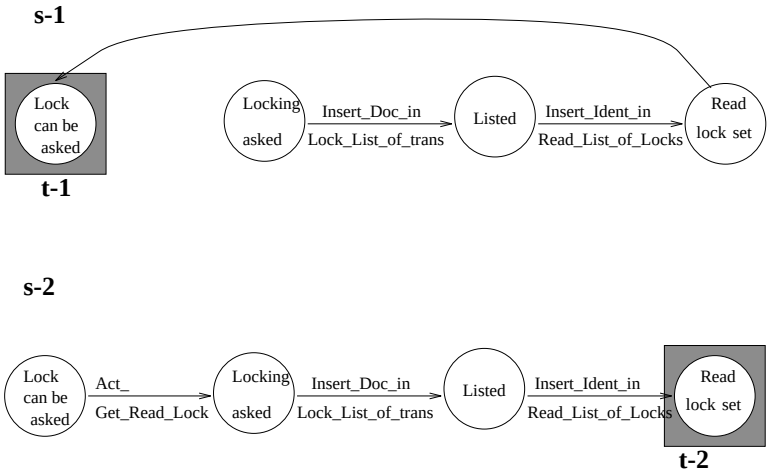


FIGURE 17.int-Get_Read_Lock's subprocesses and traps w.r.t. Locks.

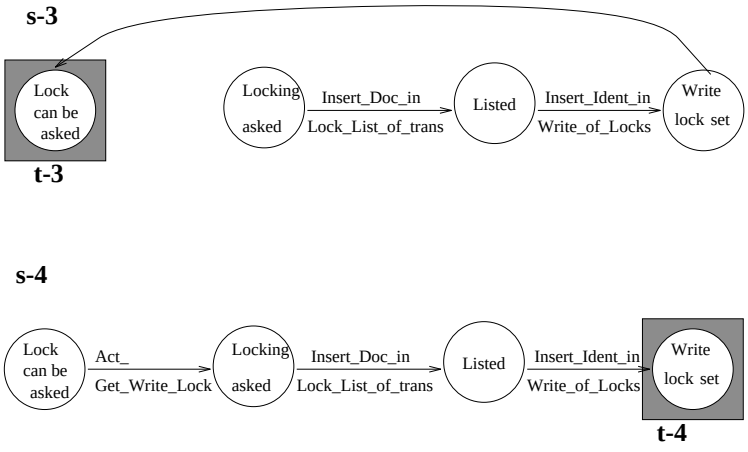


FIGURE 18.int-Get_Write_Lock's subprocesses and traps w.r.t. Locks.

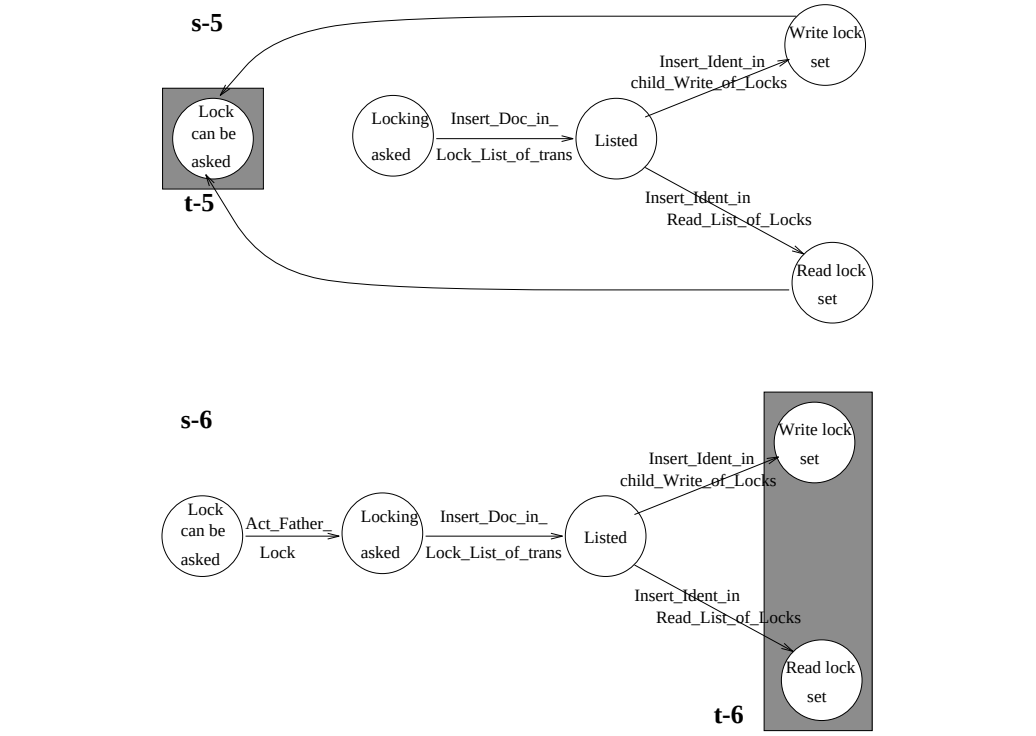


FIGURE 19.int-Get_Lock_Father's subprocesses and traps w.r.t. Locks.

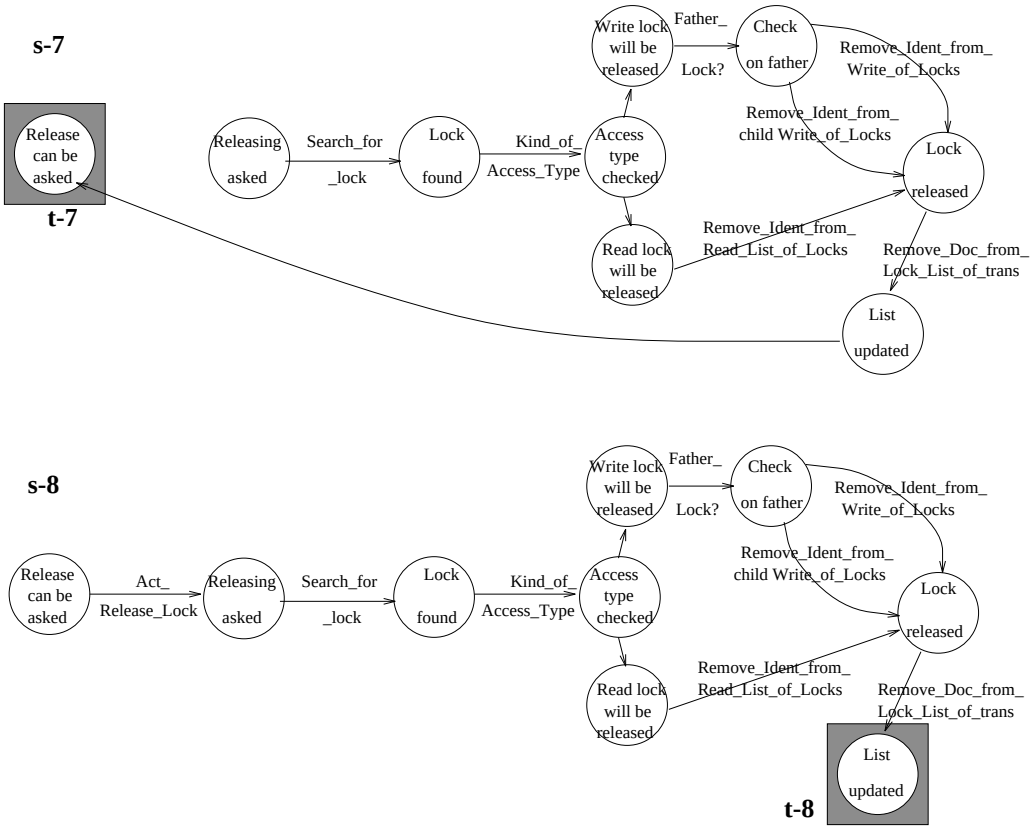


FIGURE 20.int-Release_Lock's subprocesses and traps w.r.t. Locks.

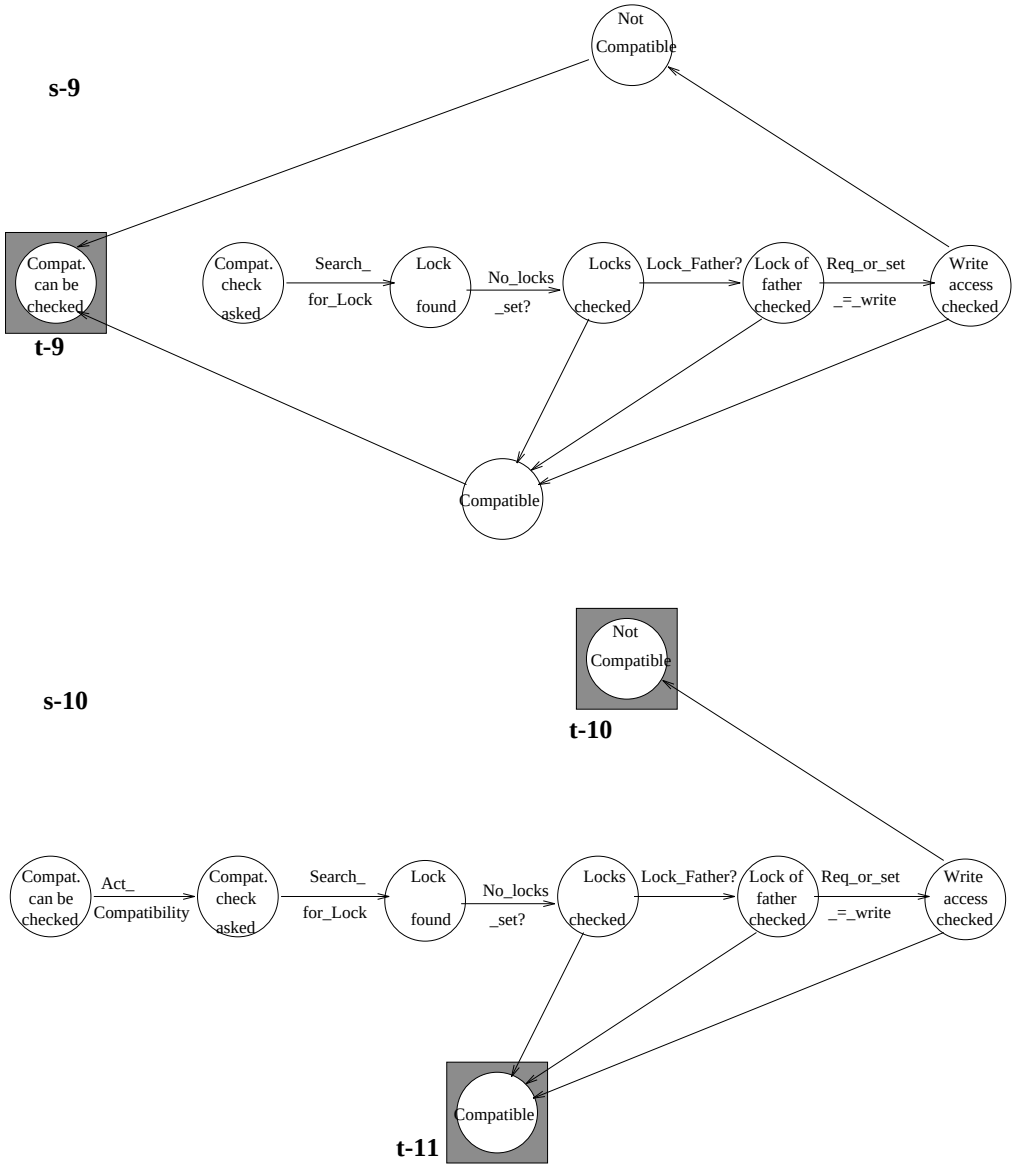
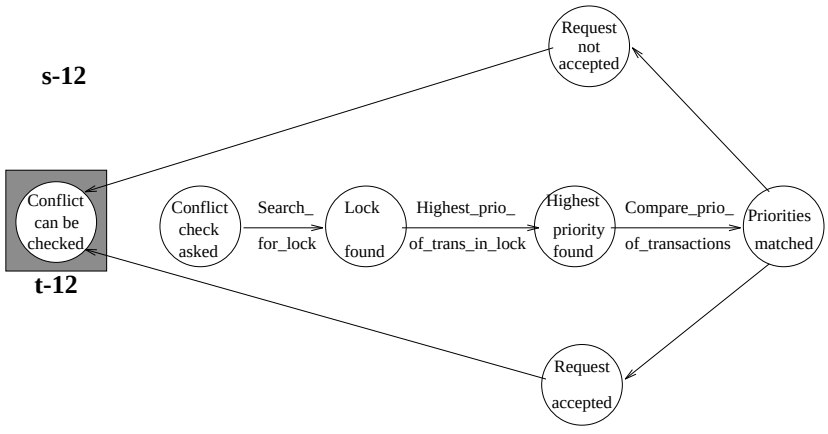


FIGURE 21.int-Compatibility's subprocesses and traps w.r.t. Locks.



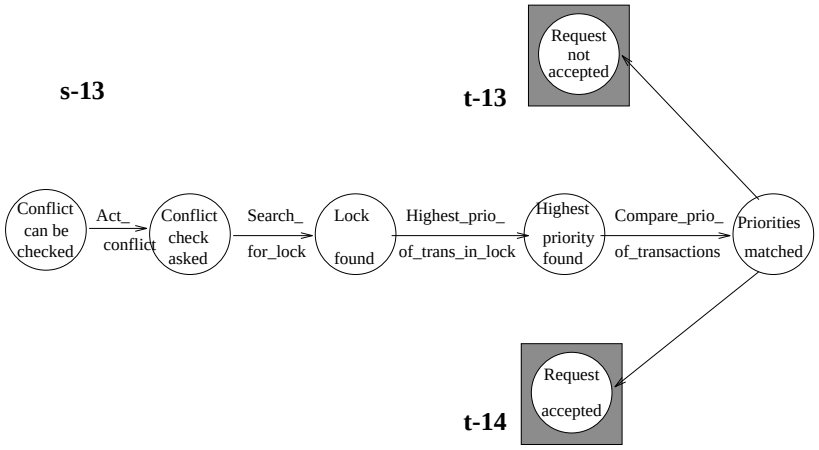
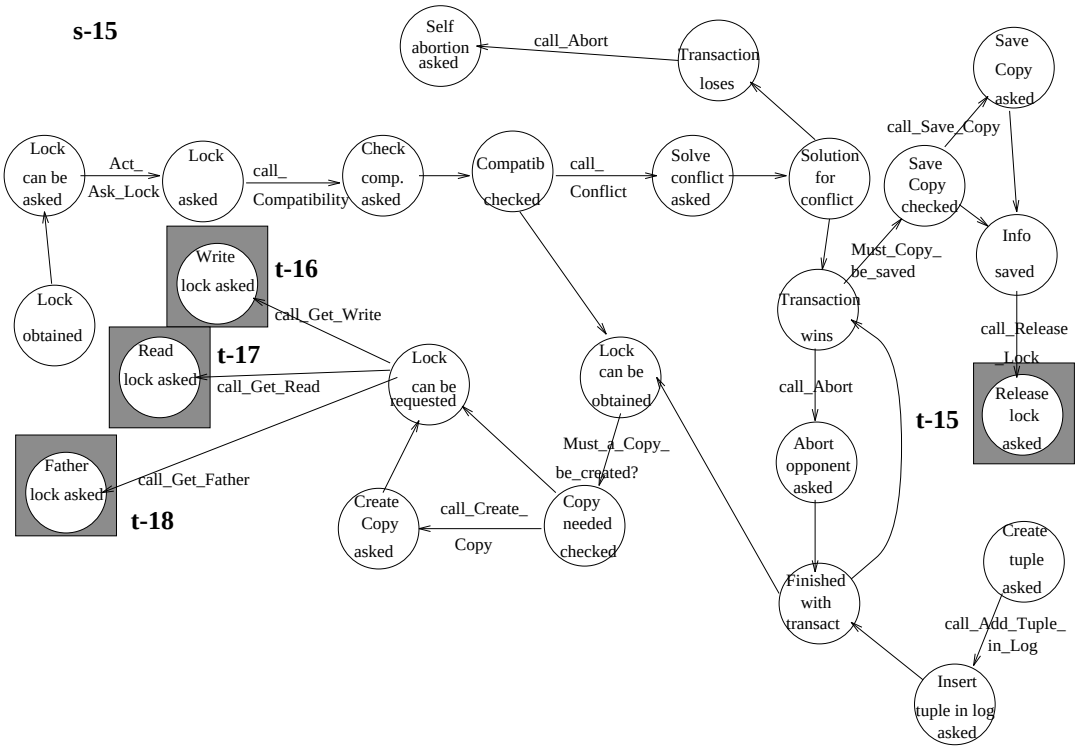


FIGURE 22.int-Conflict's subprocesses and traps w.r.t. Locks.



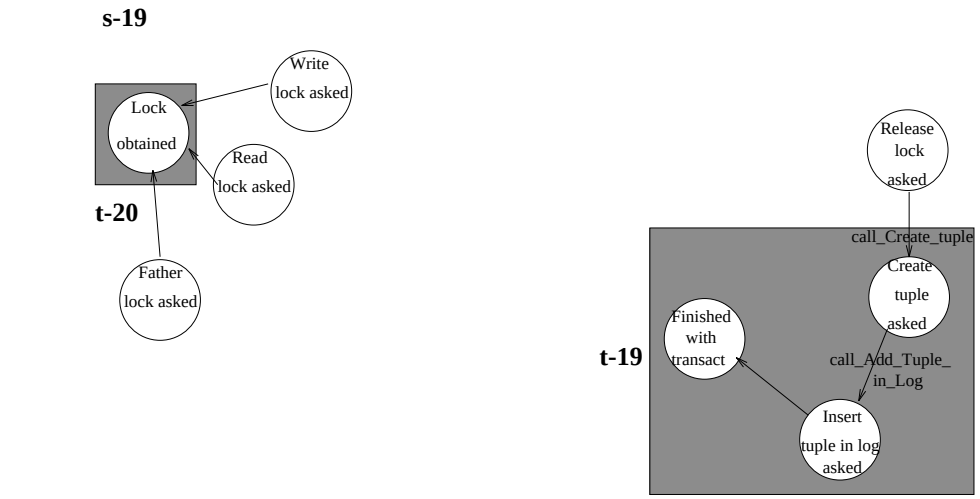


FIGURE 23.int-Ask_Lock's subprocesses and traps w.r.t. Locks.

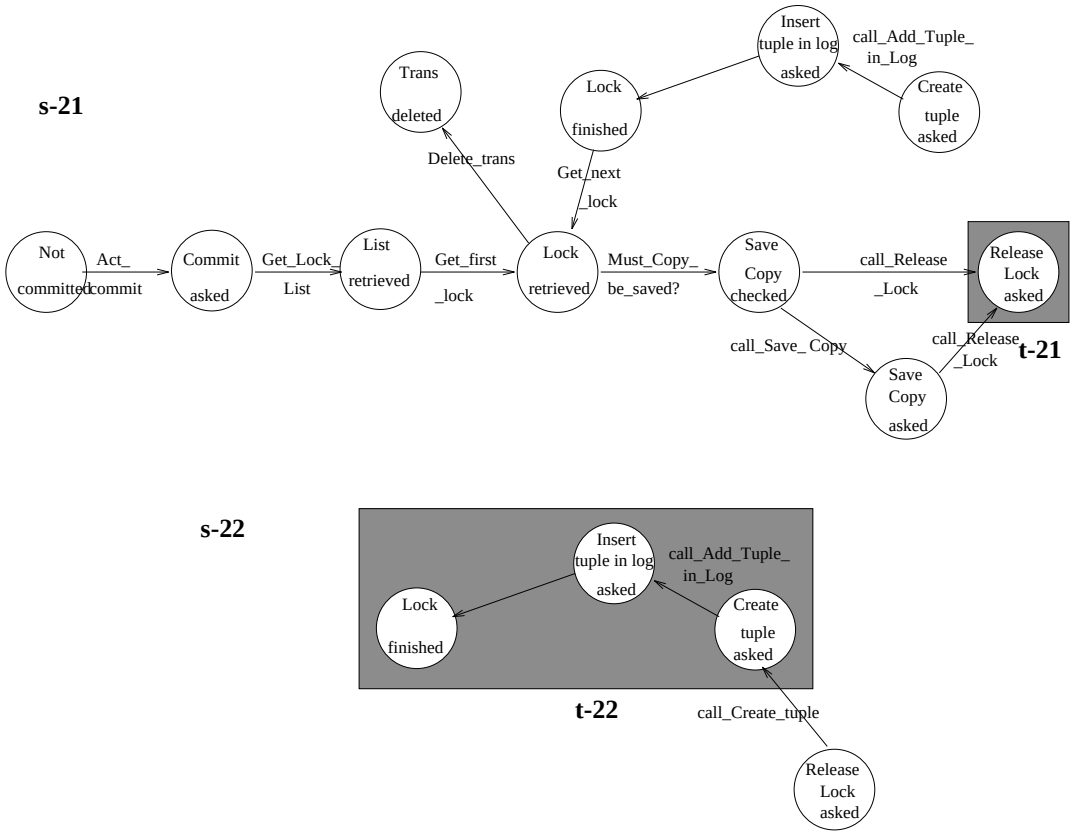


FIGURE 24.int-Commit's subprocesses and traps w.r.t. Locks.

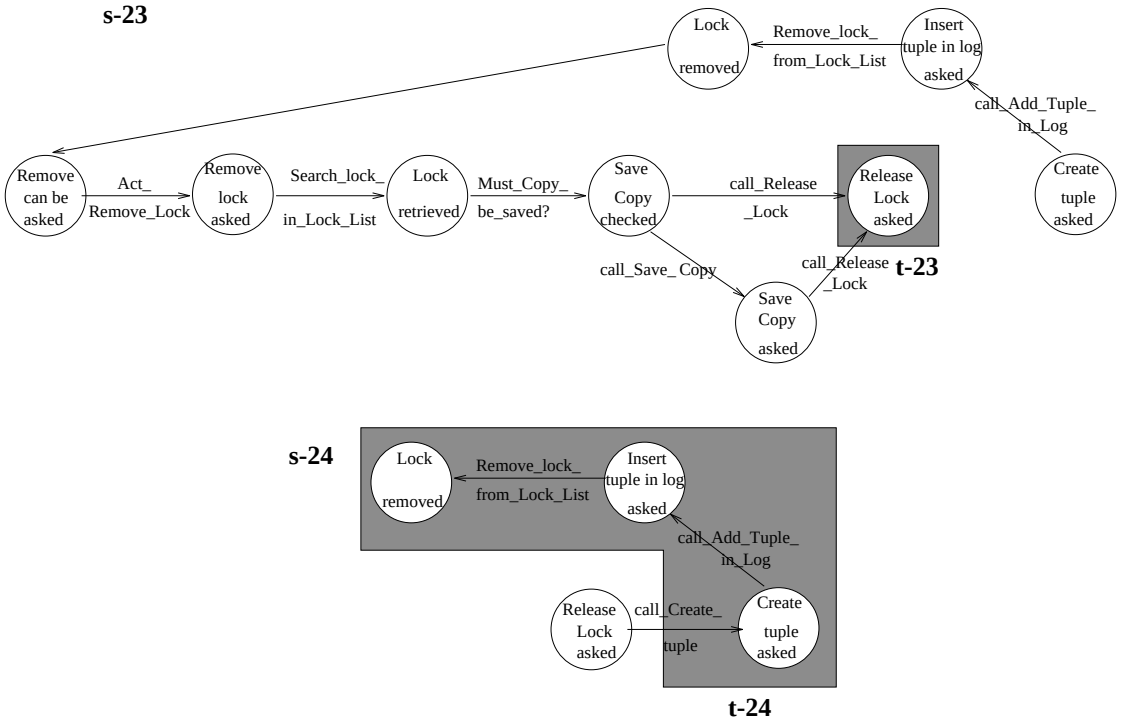


FIGURE 25.int-Remove_Lock's subprocesses and traps w.r.t. Locks.

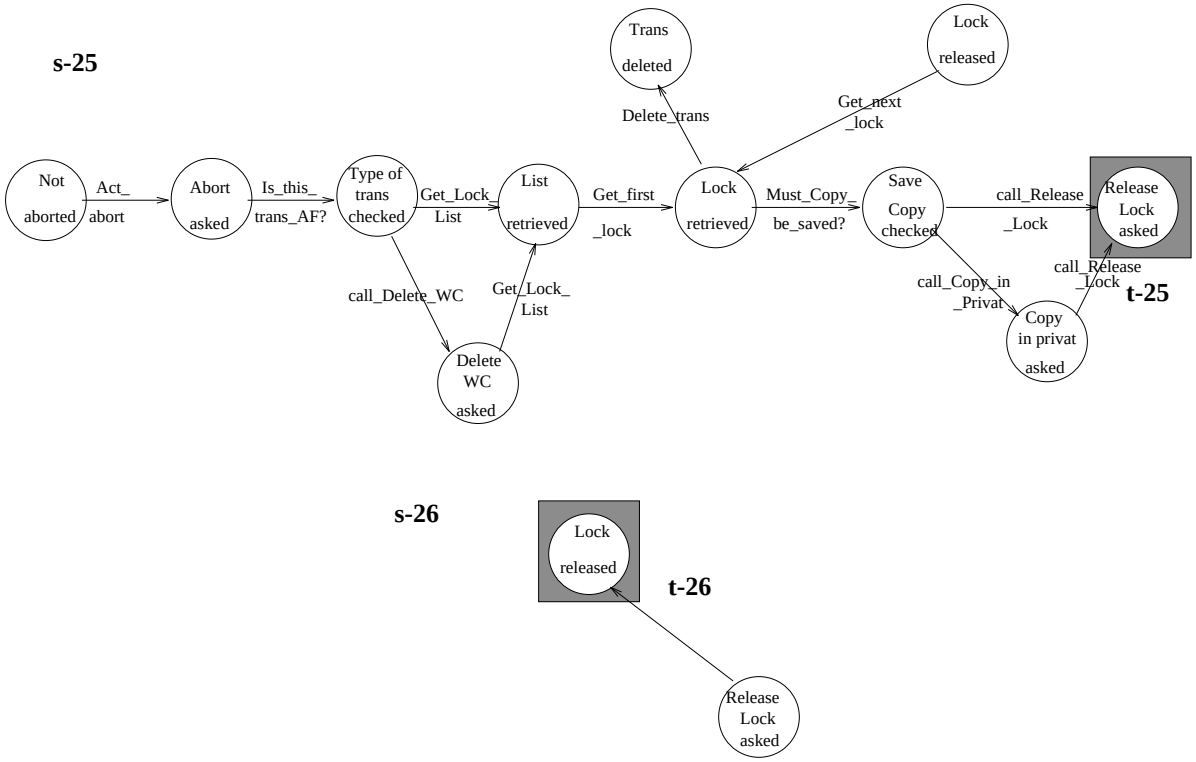


FIGURE 26.int-Abort's subprocesses and traps w.r.t. Locks.

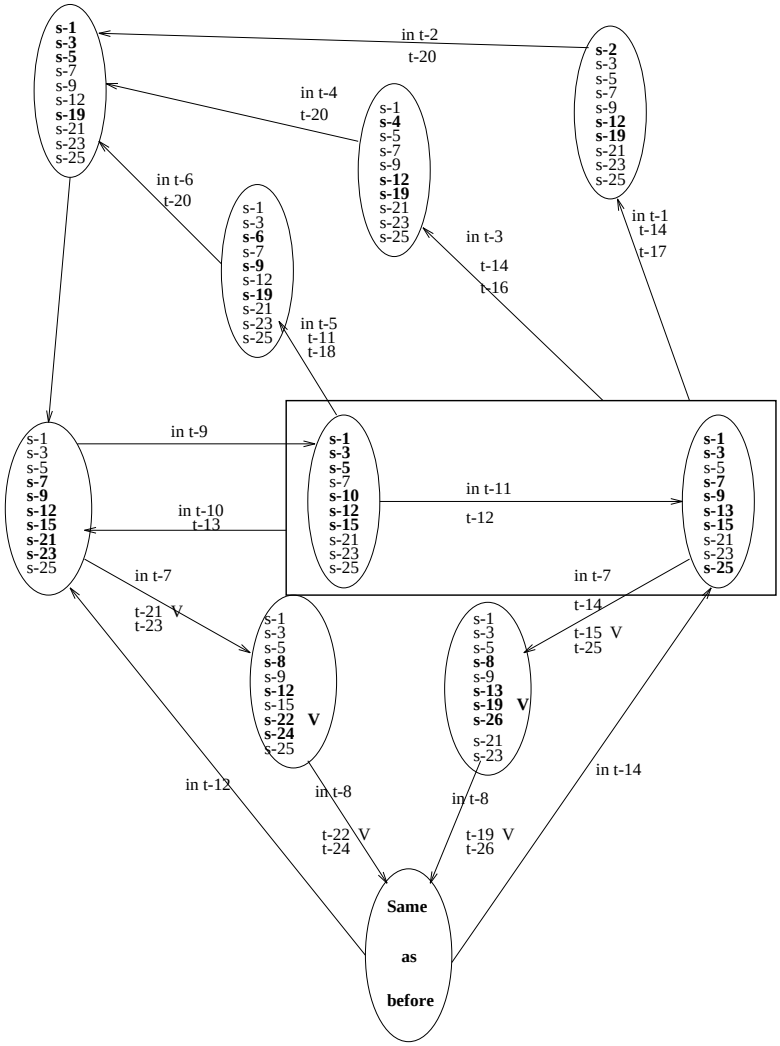


FIGURE 27. Locks, mamager of ten employees.

The next manager will be Log. Its employees are:

Add_Tuple_in_Log
Conflict_with_Log?
Commit
Remove_Lock
Update_Lock
Validate

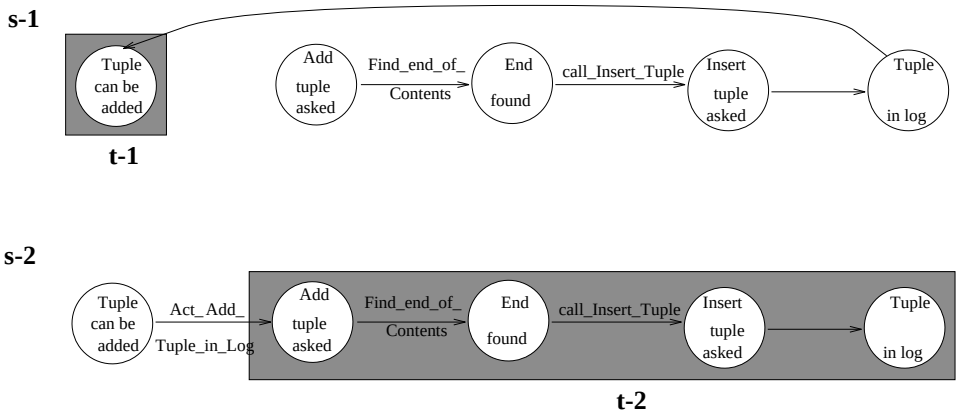
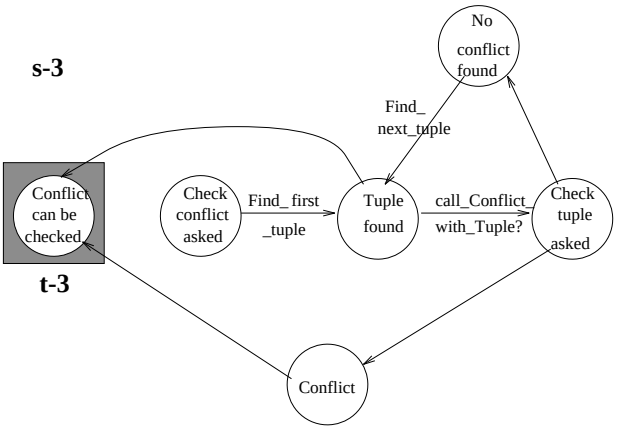


FIGURE 28.int-Add_Tuple_in_Log's subprocesses and traps w.r.t. Log.



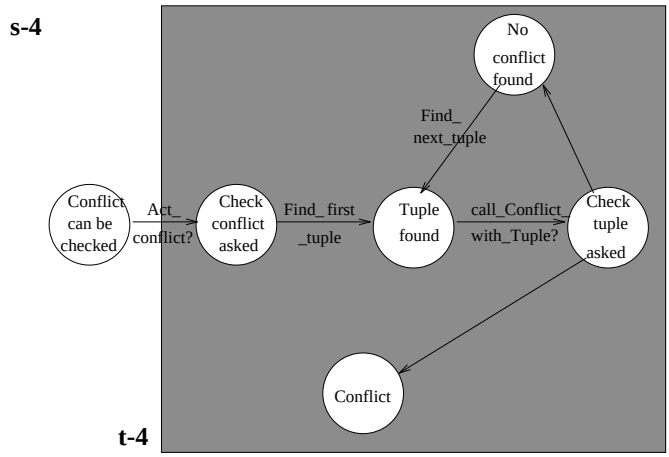
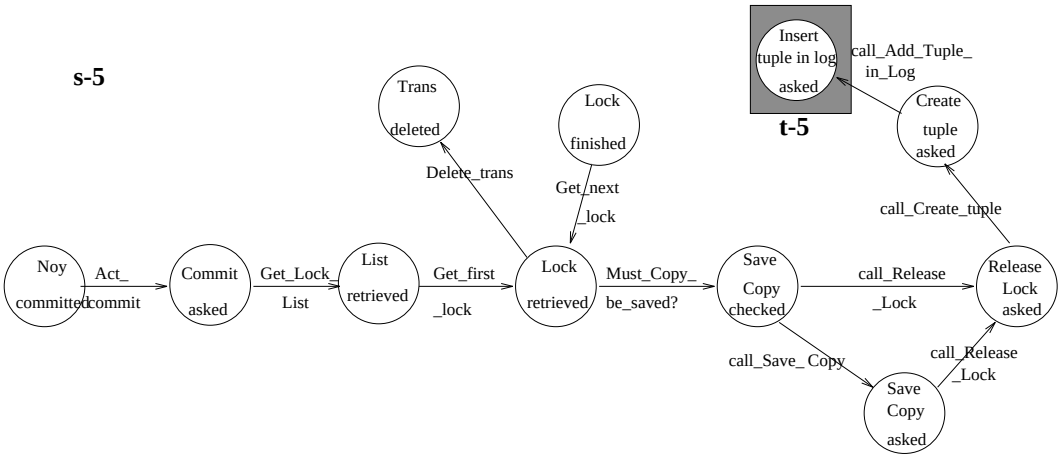


FIGURE 29.int-Conflict_with_Log?'s subprocesses and traps w.r.t. Log.



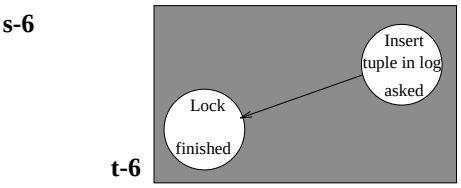
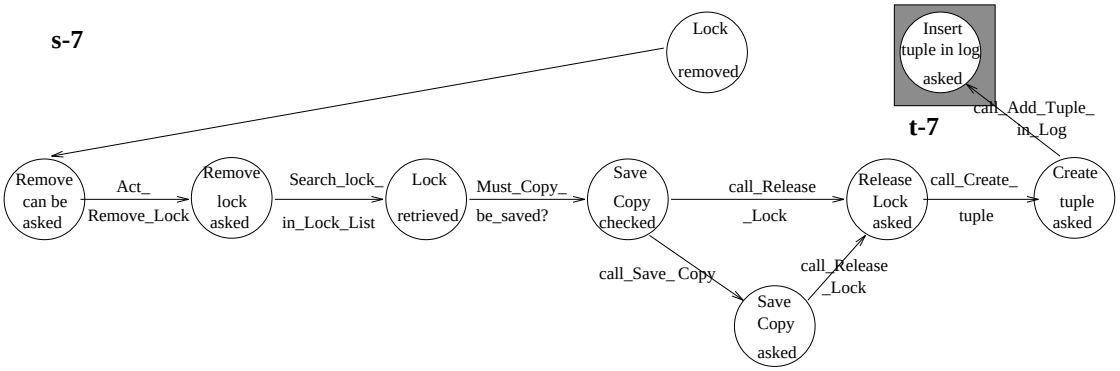


FIGURE 30.int-Commit's subprocesses and traps w.r.t. Log.



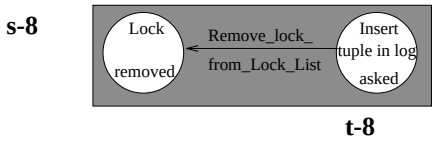
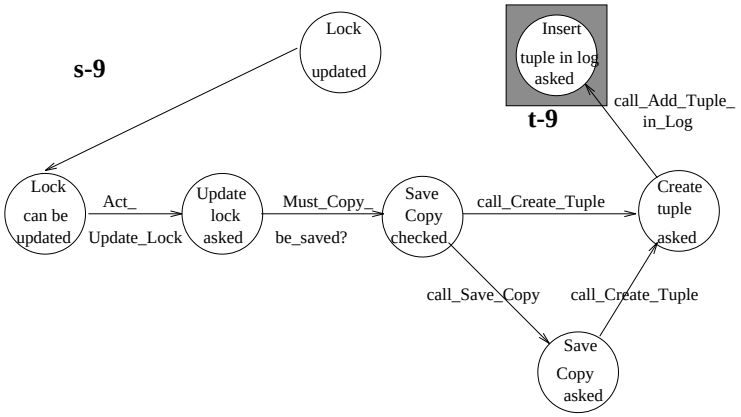


FIGURE 31.int-Remove_Lock's subprocesses and traps w.r.t. Log.



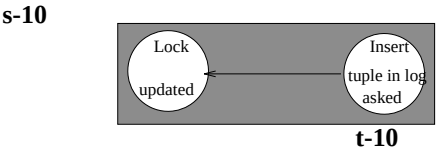


FIGURE 32.int-Update_Lock's subprocesses and traps w.r.t. Log.

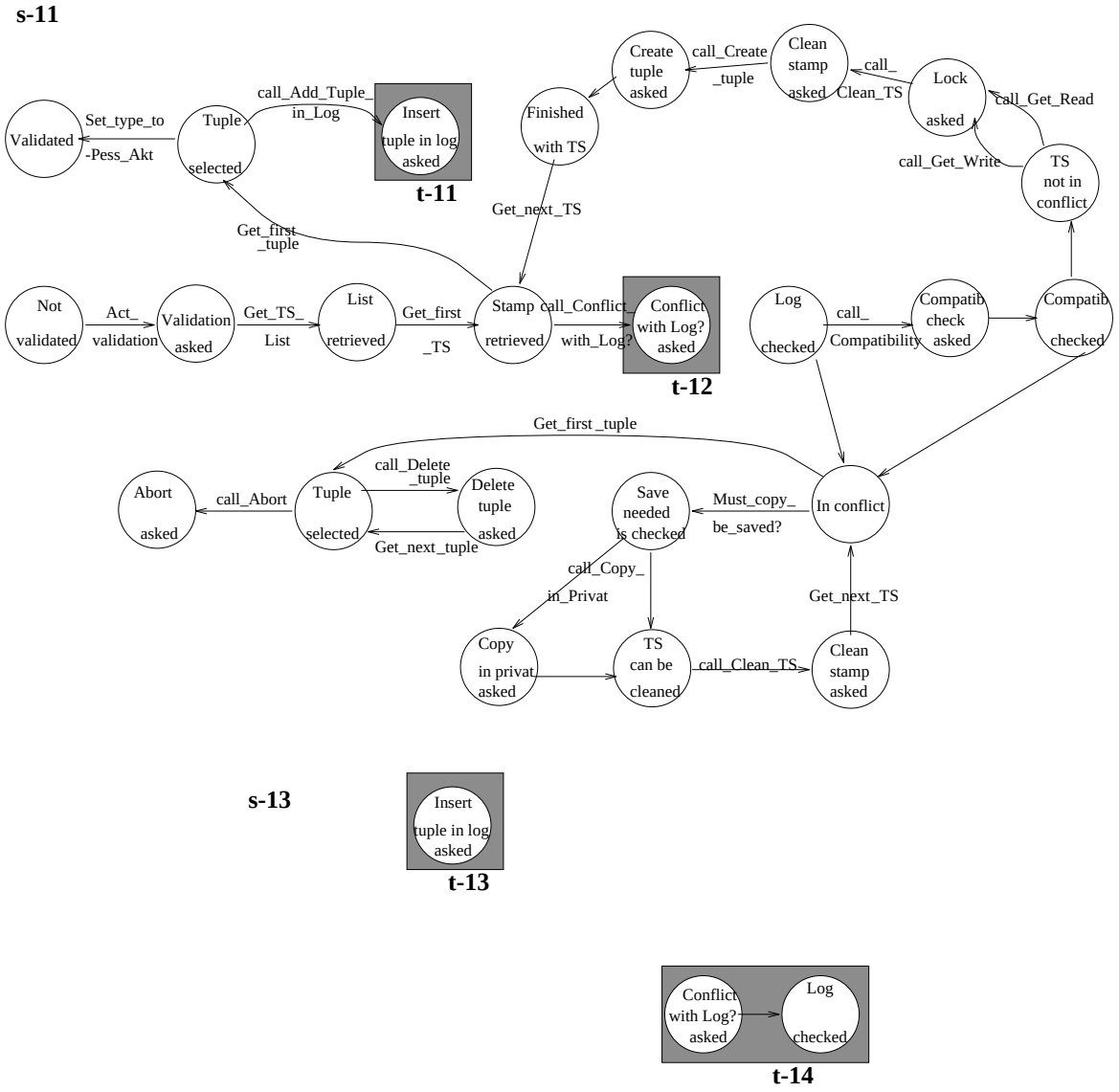


FIGURE 33.int-Validate's subprocesses and traps w.r.t. Log.

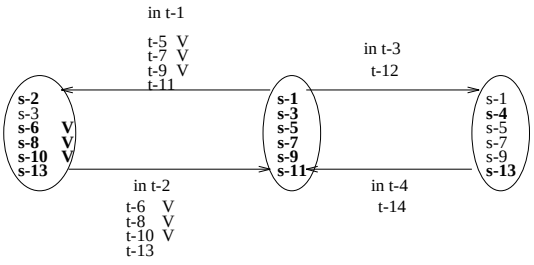


FIGURE 34. Log, mamager of six employees.

The next manager is Status. Its employees are:

New_Status
Stop_Act

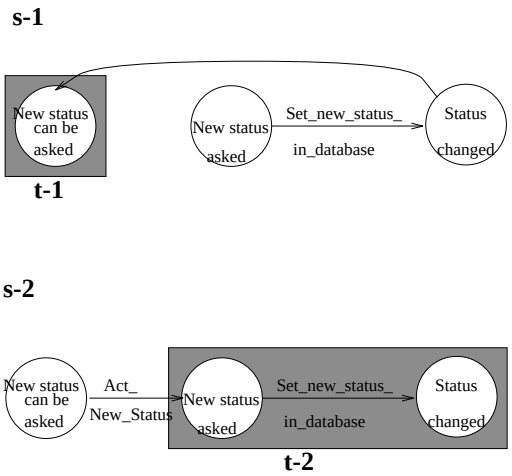
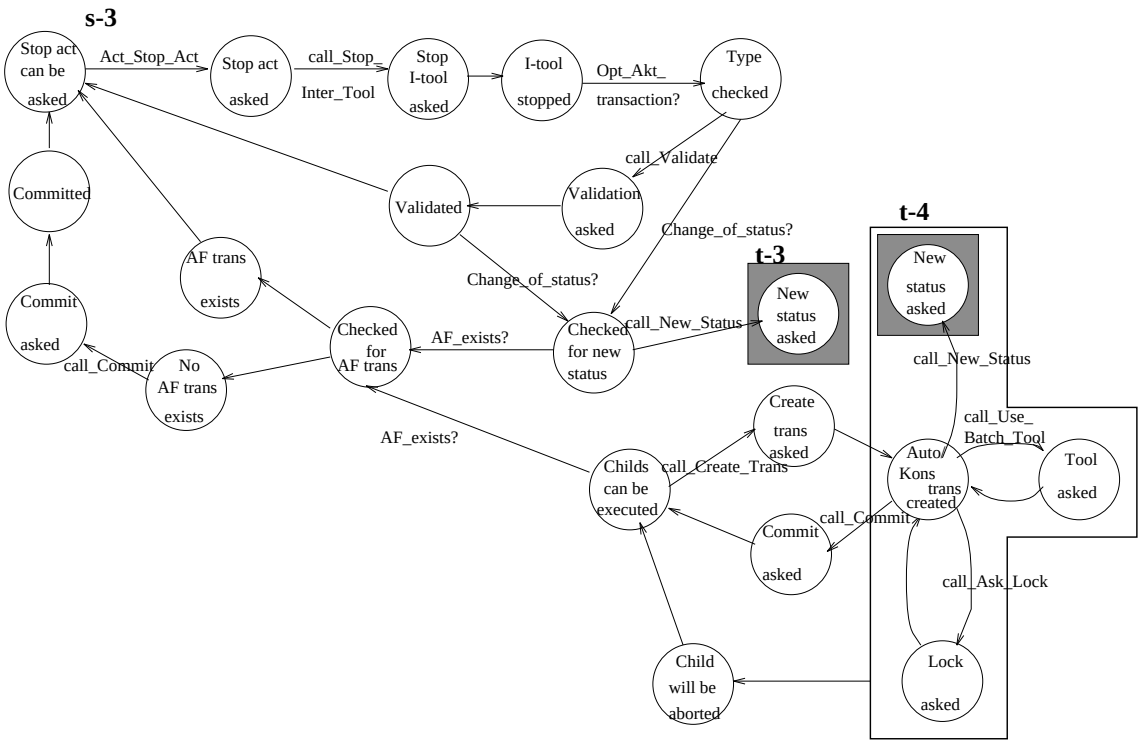


FIGURE 35.int-New_Status's subprocesses and traps w.r.t. Status.



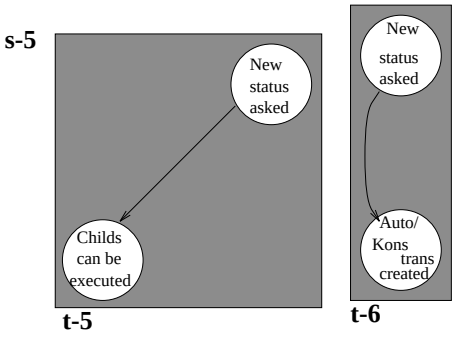


FIGURE 36.int-Stop_Act's subprocesses and traps w.r.t. Status.

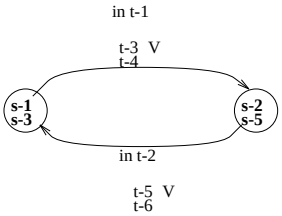


FIGURE 37.Status, manager of two employees.

The next manager is Contents. The employees of this manager are:

Start_Inter_Tool
Stop_Inter_Tool
Use_Batch_Tool
Save_Copy
Create_Copy
Copy_in_Privat
Start_Act
Stop_Act
Ask_Lock
Set_TS
Abort
Validate
Commit
Remove_Lock
Update_Lock

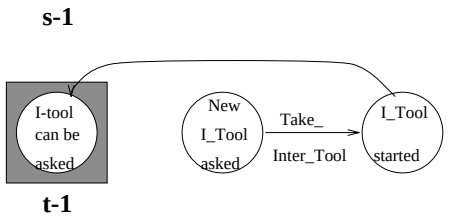


FIGURE 38.int-Start_Inter_Tool's subprocesses and traps w.r.t. Contents.

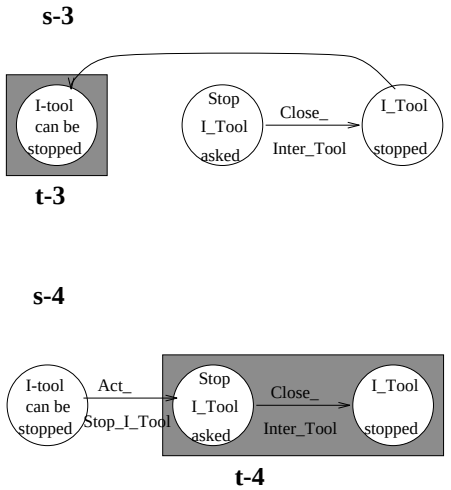


FIGURE 39.int-Stop_Inter_Tool's subprocesses and traps w.r.t. Contents.

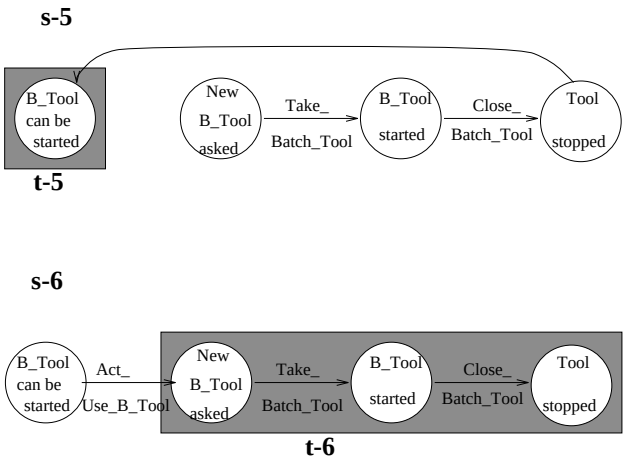


FIGURE 40.int-Use_Batch_Tool's subprocesses and traps w.r.t. Contents.

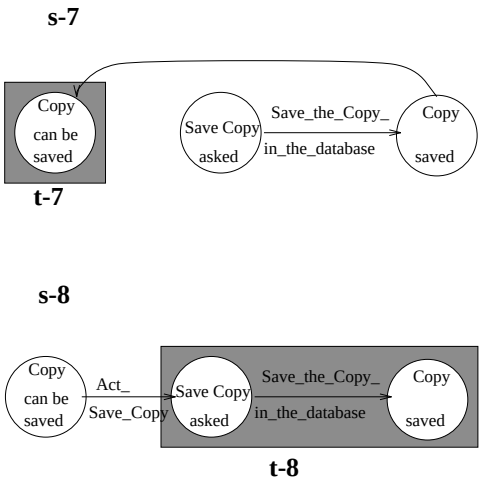


FIGURE 41.int-Save_Copy's subprocesses and traps w.r.t. Contents.

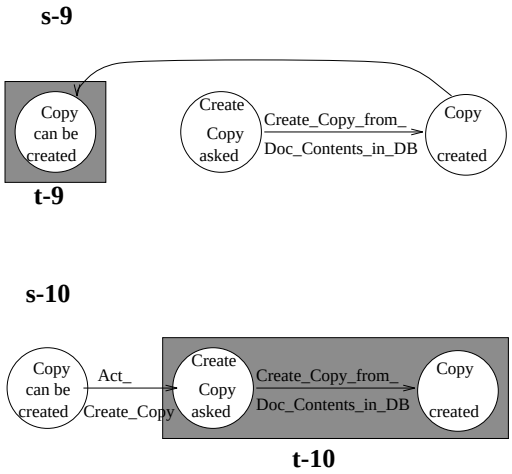


FIGURE 42.int-Create_Copy's subprocesses and traps w.r.t. Contents.

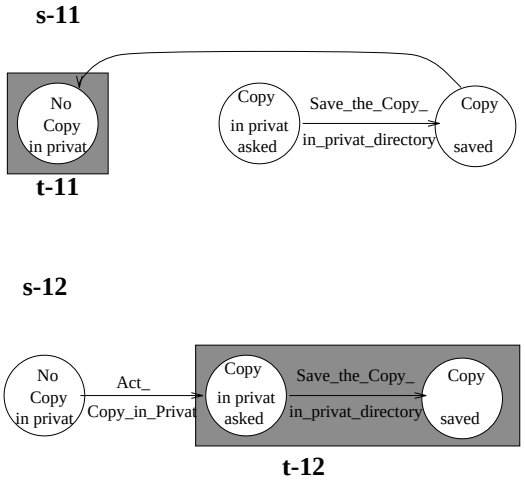


FIGURE 43.int-Copy_in_Privat's subprocesses and traps w.r.t. Contents.

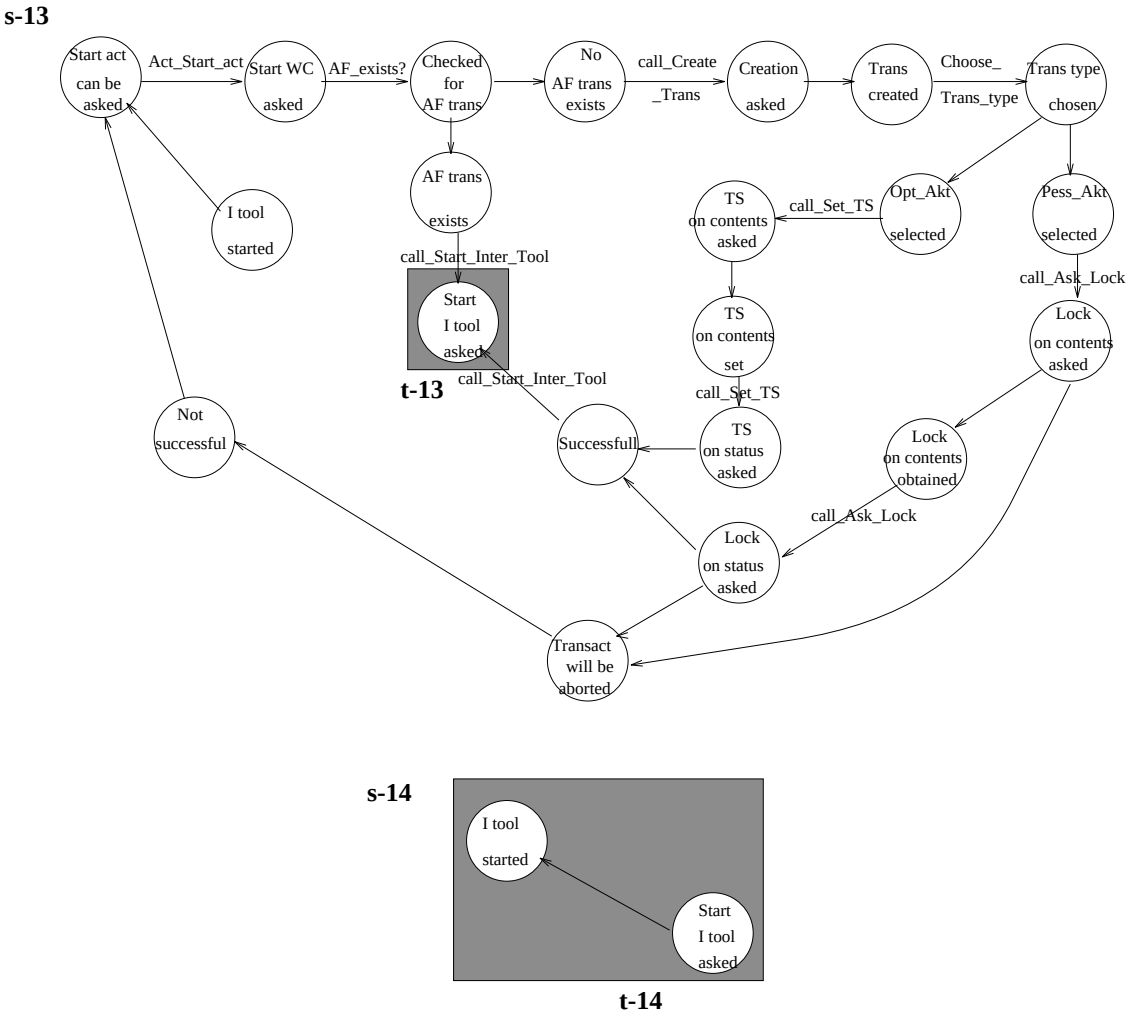
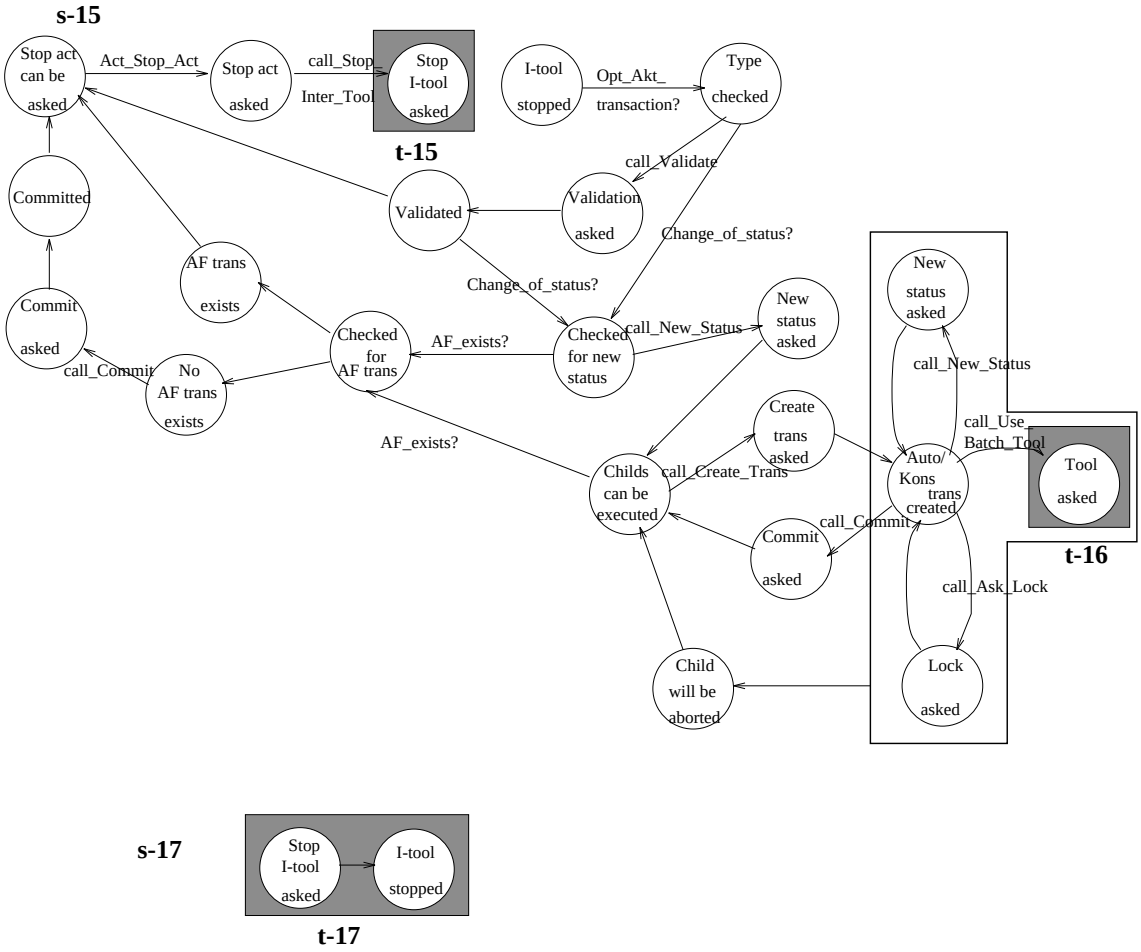


FIGURE 44.int-Start_Act's subprocesses and traps w.r.t. Contents.



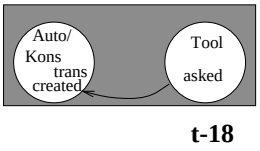


FIGURE 45.int-Stop_Act's subprocesses and traps w.r.t. Contents.

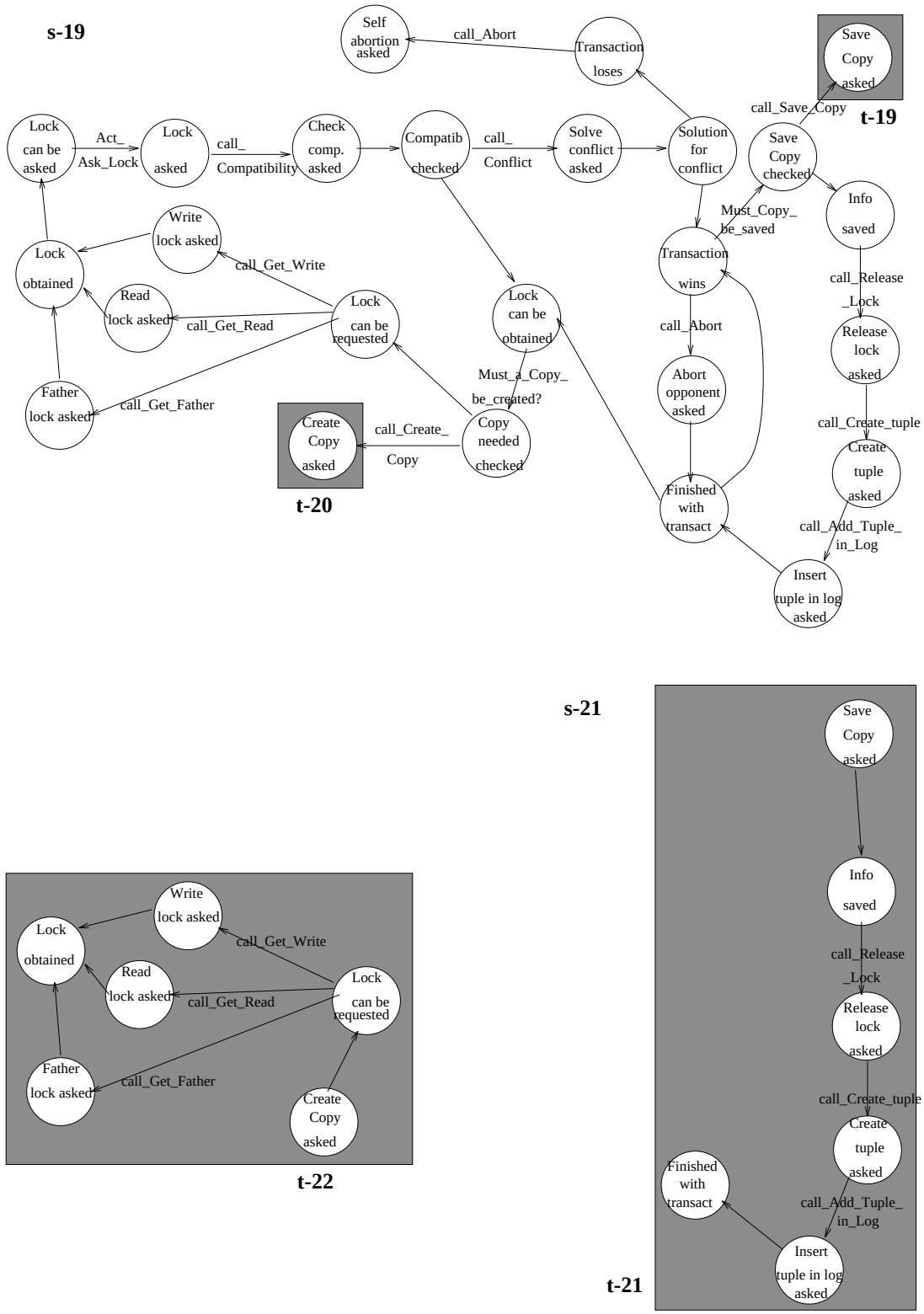


FIGURE 46.int-Ask_Lock's subprocesses and traps w.r.t. Contents.

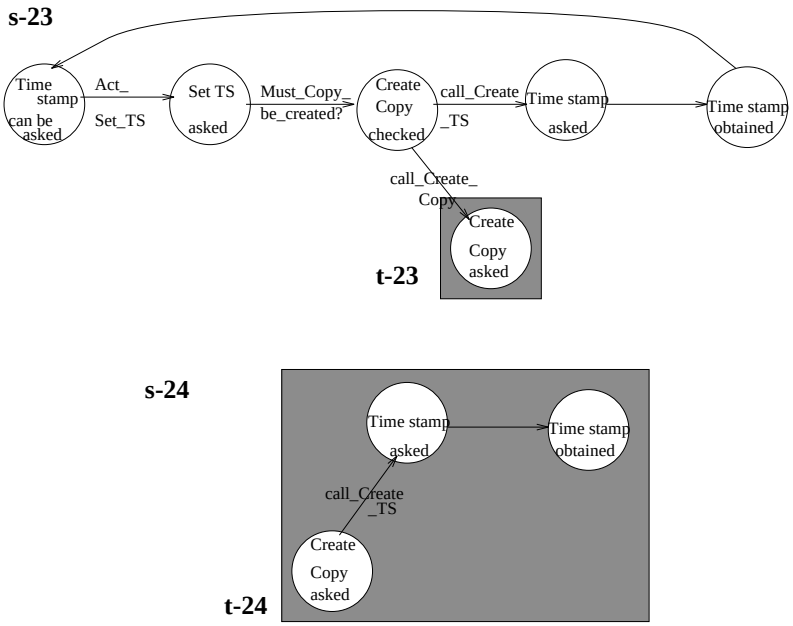


FIGURE 47.int-Set_TS's subprocesses and traps w.r.t. Contents.

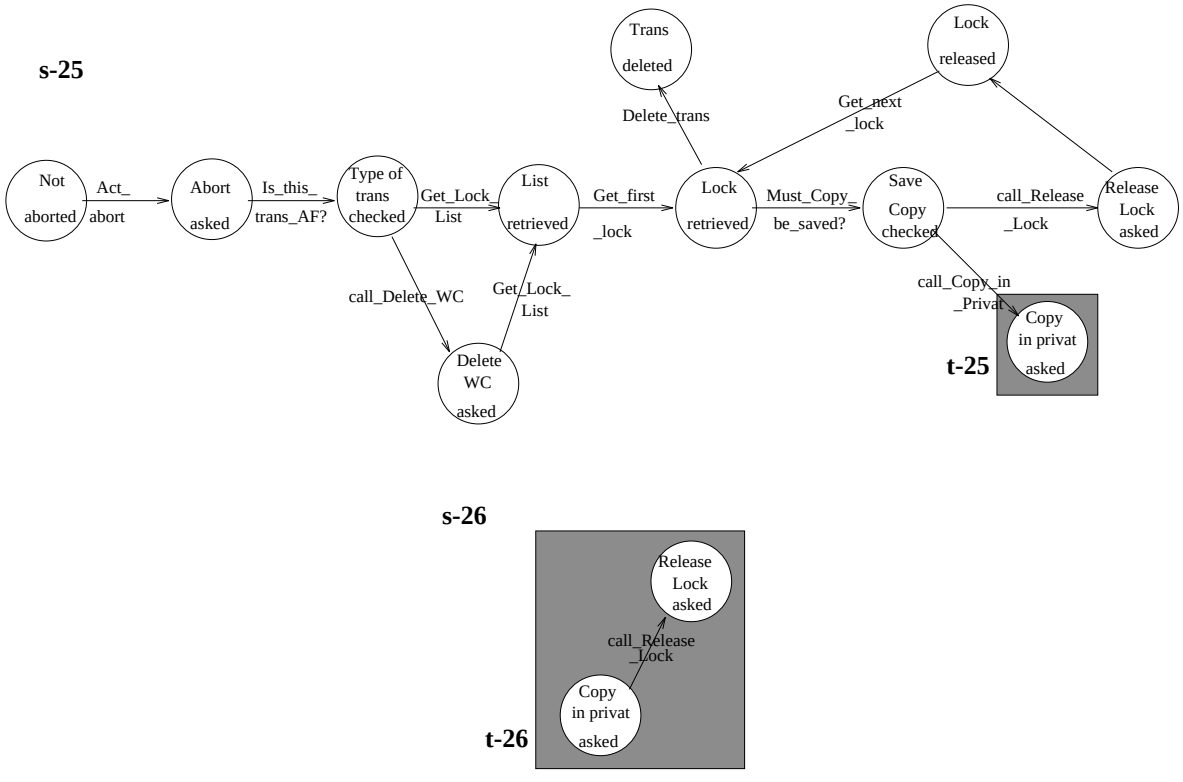


FIGURE 48.int-Abort's subprocesses and traps w.r.t. Contents.

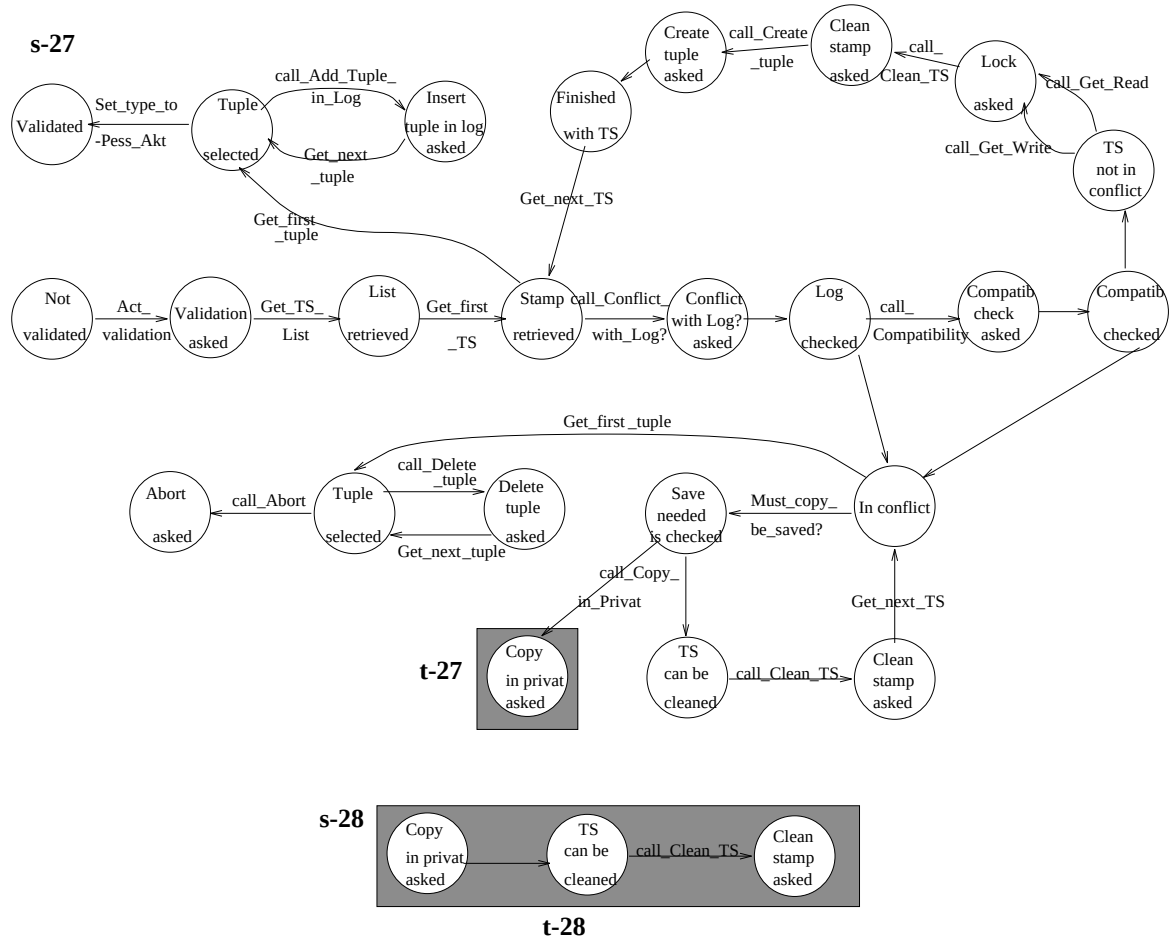
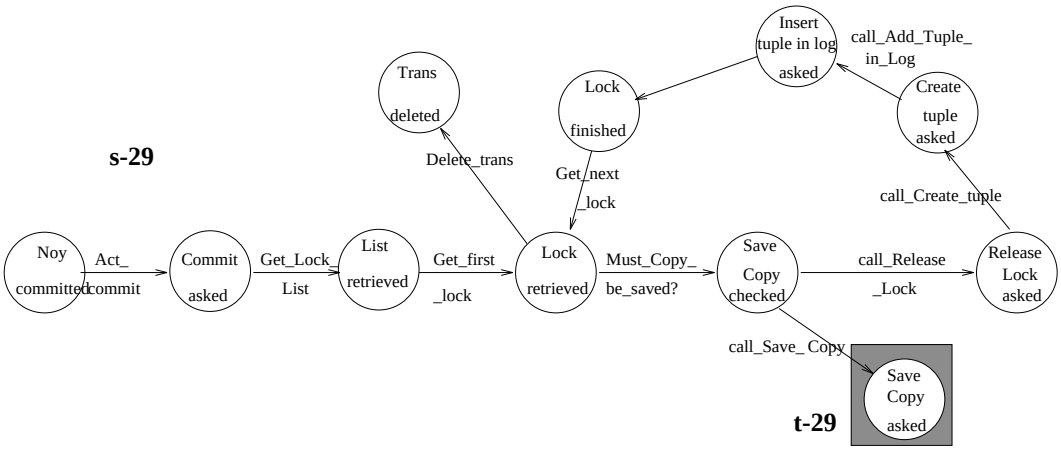


FIGURE 49.int-Validate's subprocesses and traps w.r.t. Contents.



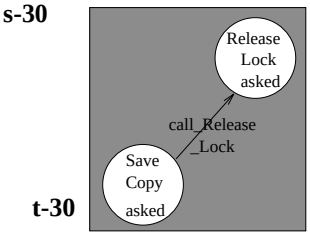
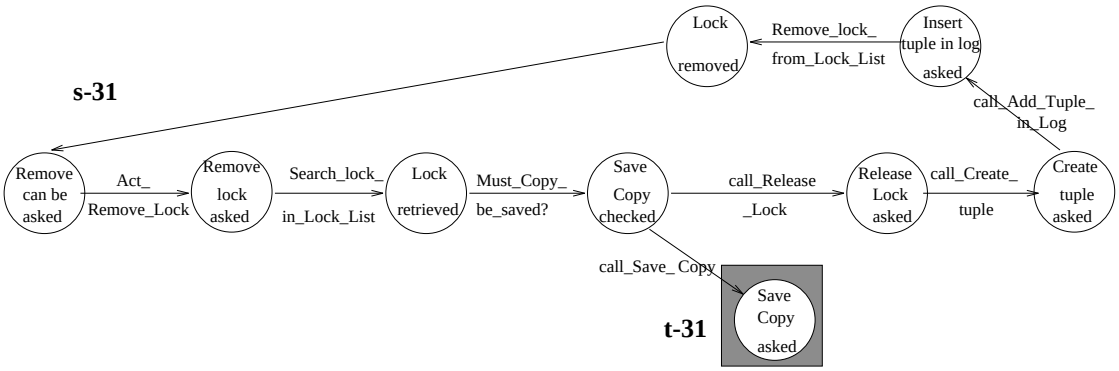


FIGURE 50.int-Commit's subprocesses and traps w.r.t. Contents.



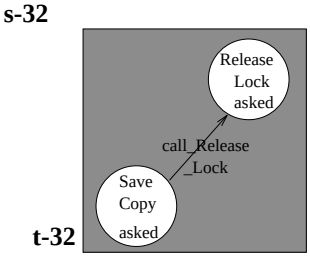
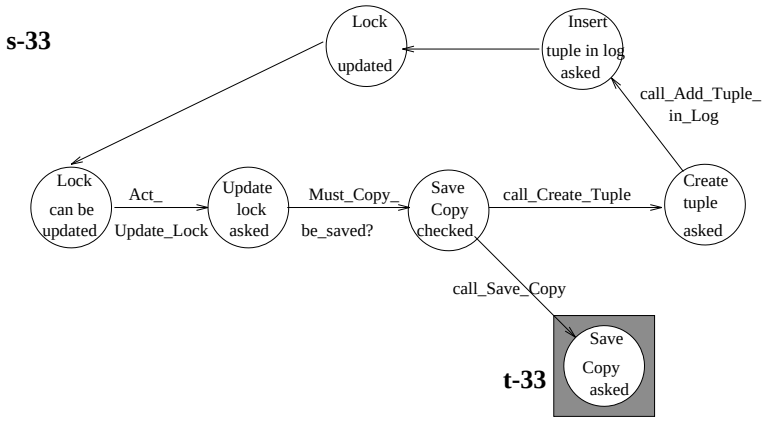


FIGURE 51.int-Remove_Lock's subprocesses and traps w.r.t. Contents.



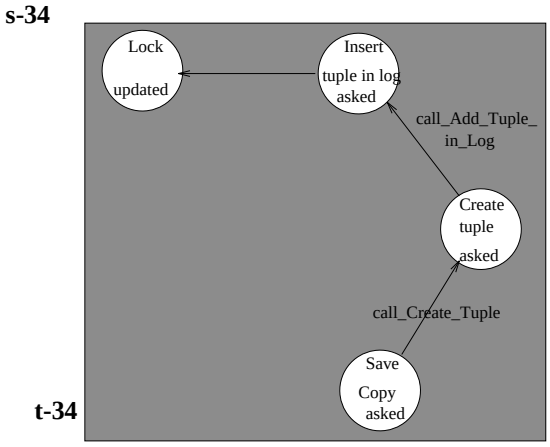


FIGURE 52.int-Update_Lock's subprocesses and traps w.r.t. Contents.

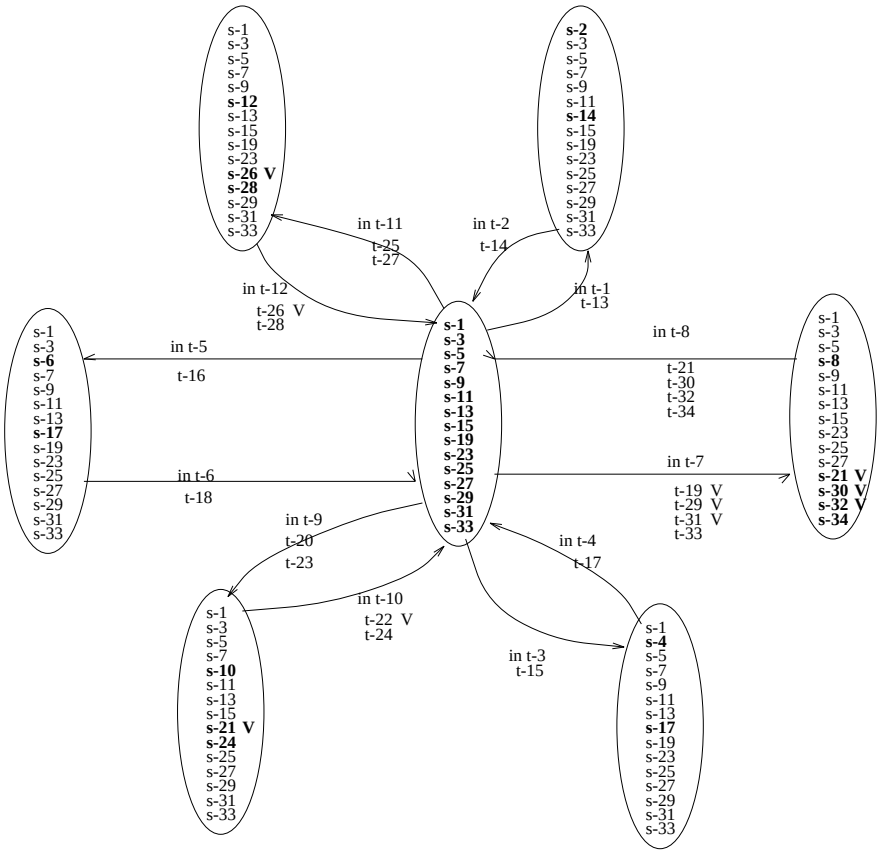


FIGURE 52. Contents, manager of fifteen employees.

Next, the five managers of the subclasses of Process Transaction will be specified. The five managers are the external behaviours of the classes Kons Transaction, Auto Transaction, Pess_AF Transaction, Pess_Akt transaction and Opt_Akt Transaction.
All the managers have the following three employees:

Create_Trans
Commit
Abort

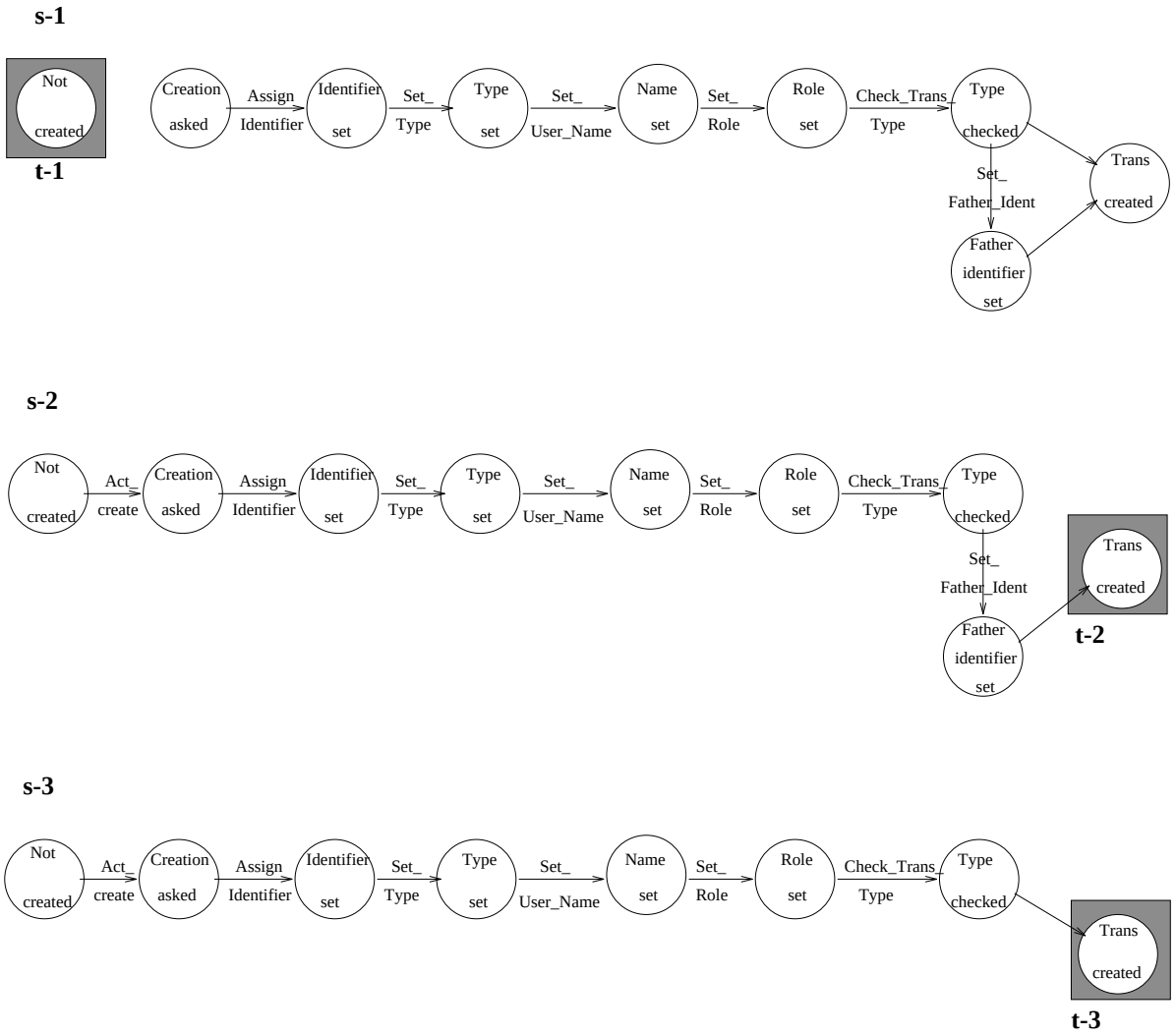


FIGURE 54.int-Create_Trans's subprocesses and traps w.r.t. Process Transaction.

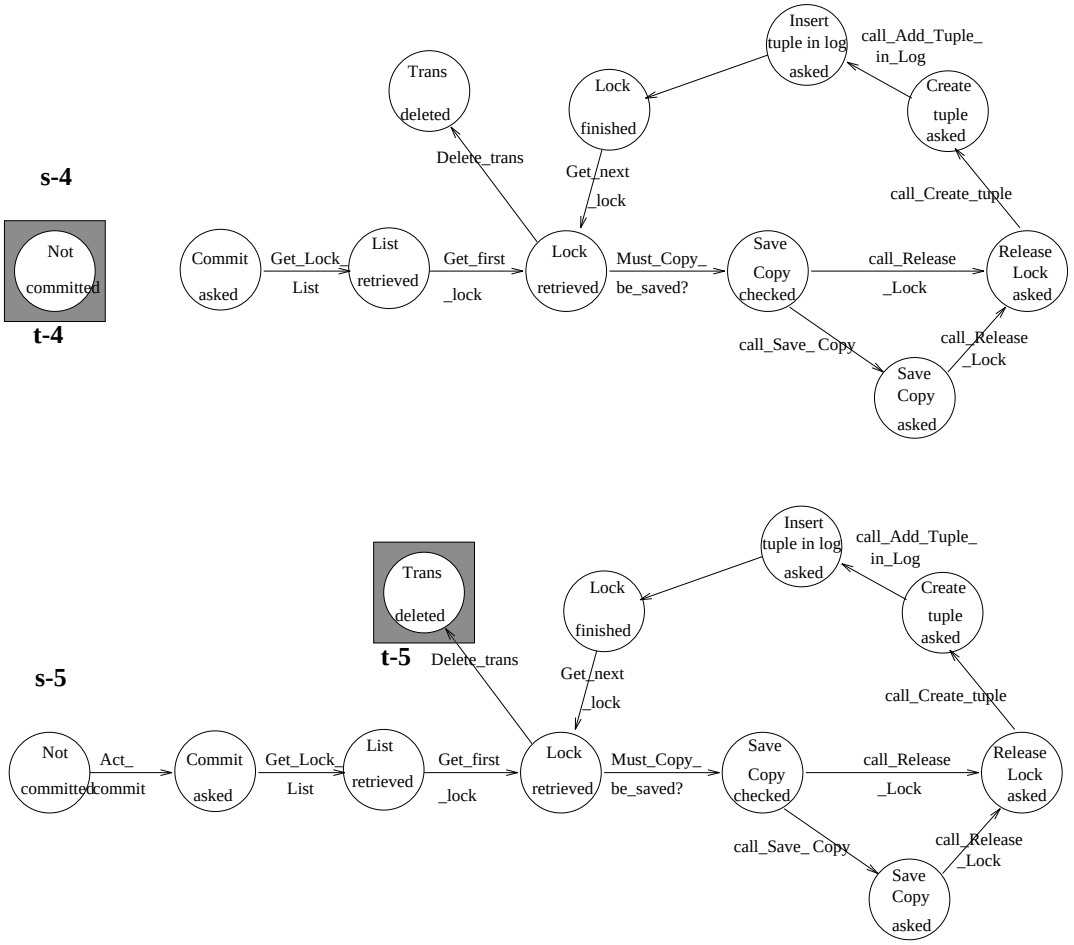
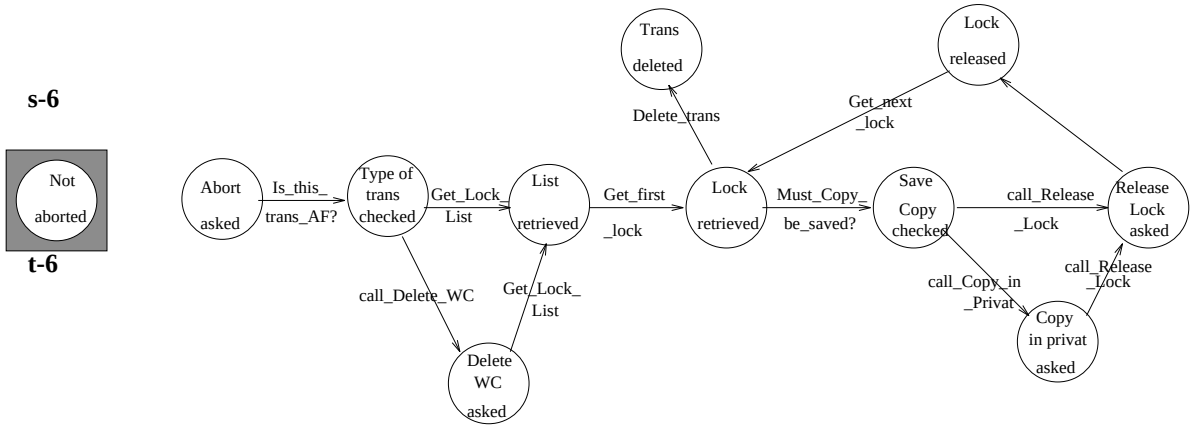


FIGURE 55.int-Commit's subprocesses and traps w.r.t. Process Transaction.



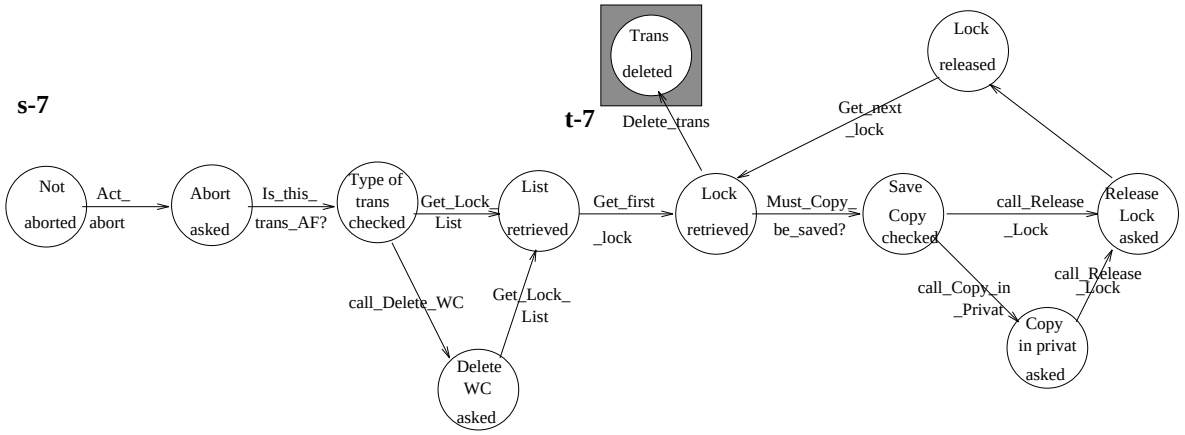
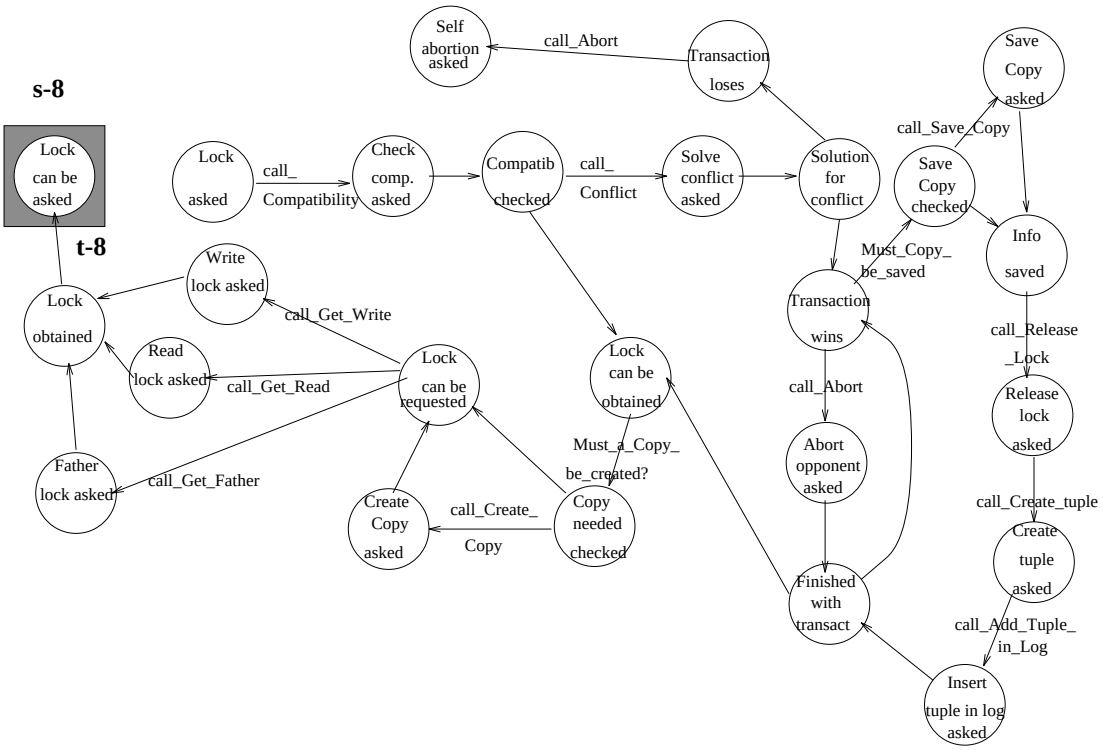


FIGURE 55.int-Abort's subprocesses and traps w.r.t. Process Transaction.

The first manager is Kons. This manager has as extra employees.

Ask_Lock
Stop_Act



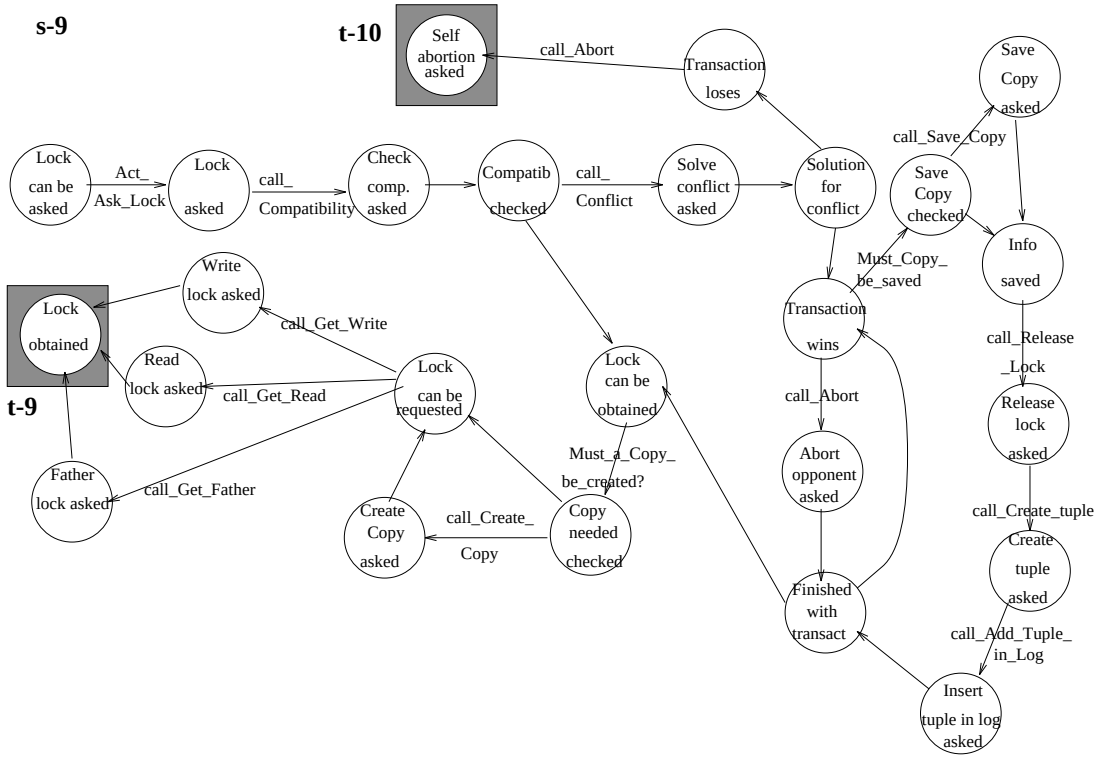
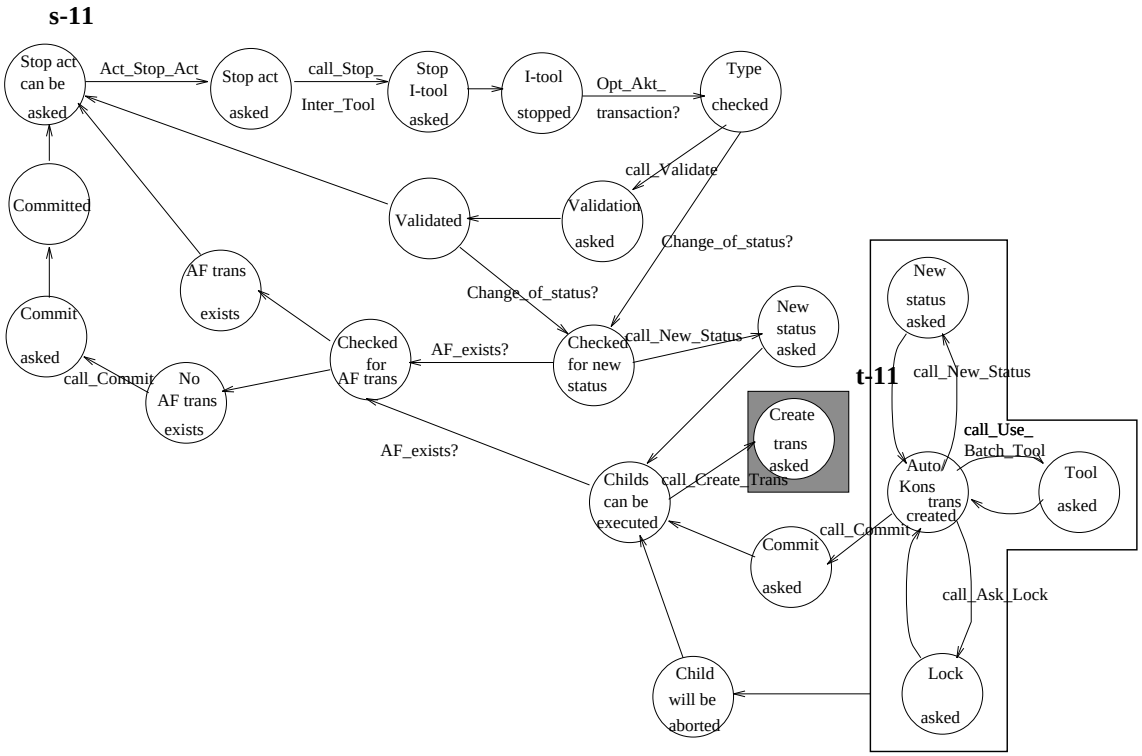
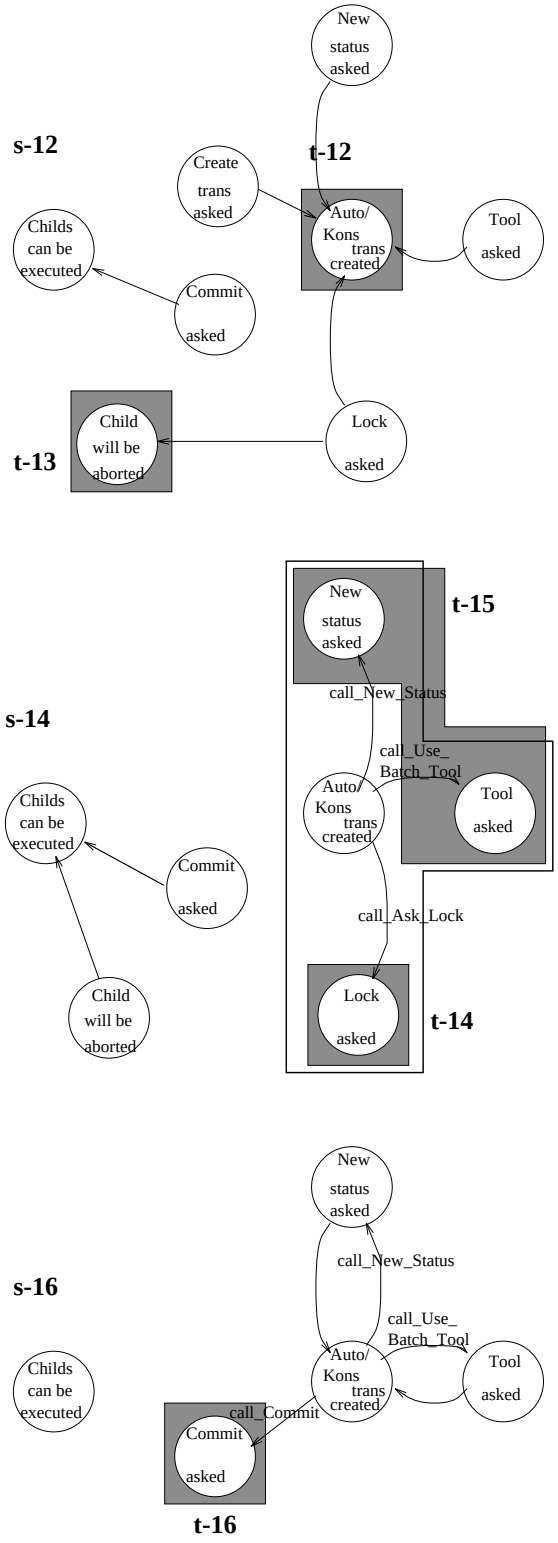


FIGURE 56.int-Ask_Lock's subprocesses and traps w.r.t. Kons Transaction.




s-12
t-12
New status asked
Create trans asked
Auto/ Kons trans created
Tool asked
Childs can be executed
Commit asked
Child will be aborted
Lock asked
t-13
s-14
t-15
New status asked
call_New_Status
call_Use_ Batch_Tool
Auto/ Kons trans created
Tool asked
Childs can be executed
Commit asked
call_Ask_Lock
Child will be aborted
Lock asked
t-14
s-16
New status asked
call_New_Status
call_Use_ Batch_Tool
Childs can be executed
Auto/ Kons trans created
Tool asked
call_Commit
Commit asked
t-16

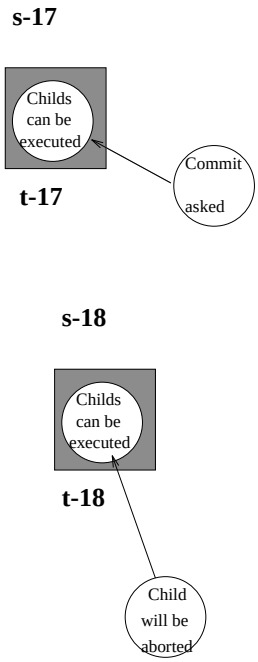


FIGURE 55.int-Stop_Act's subprocesses and traps w.r.t. Kons Transaction.

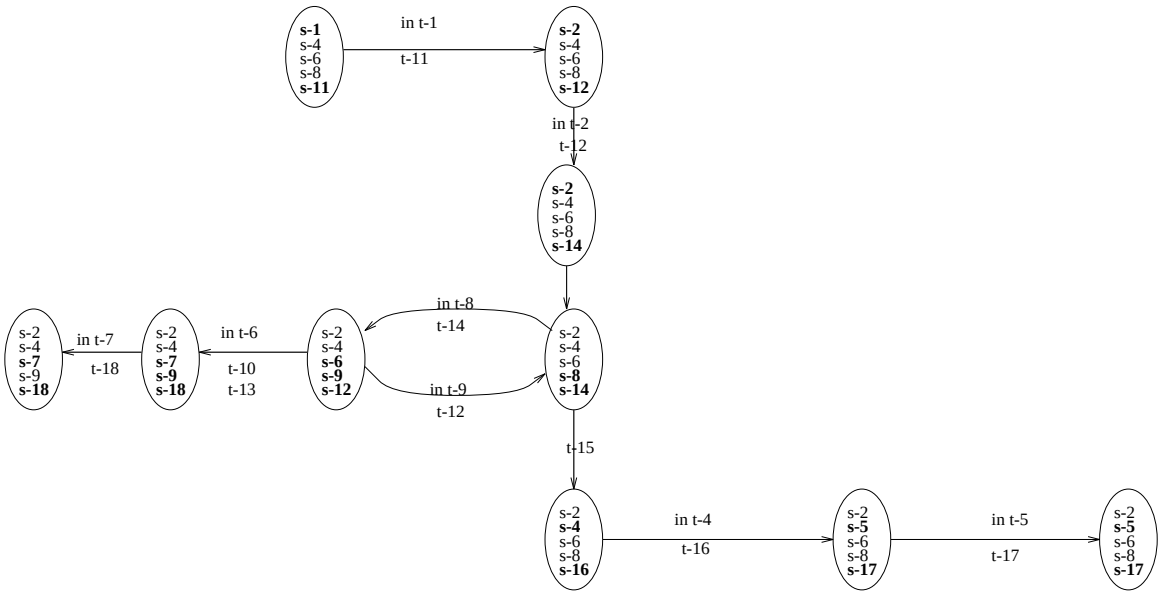


FIGURE 56. Kons Transaction, manager of five employees.

The next manager is Auto. Extra employees are:

Ask_Lock (same instance)

Stop_Act

Ask_Lock (other instance)

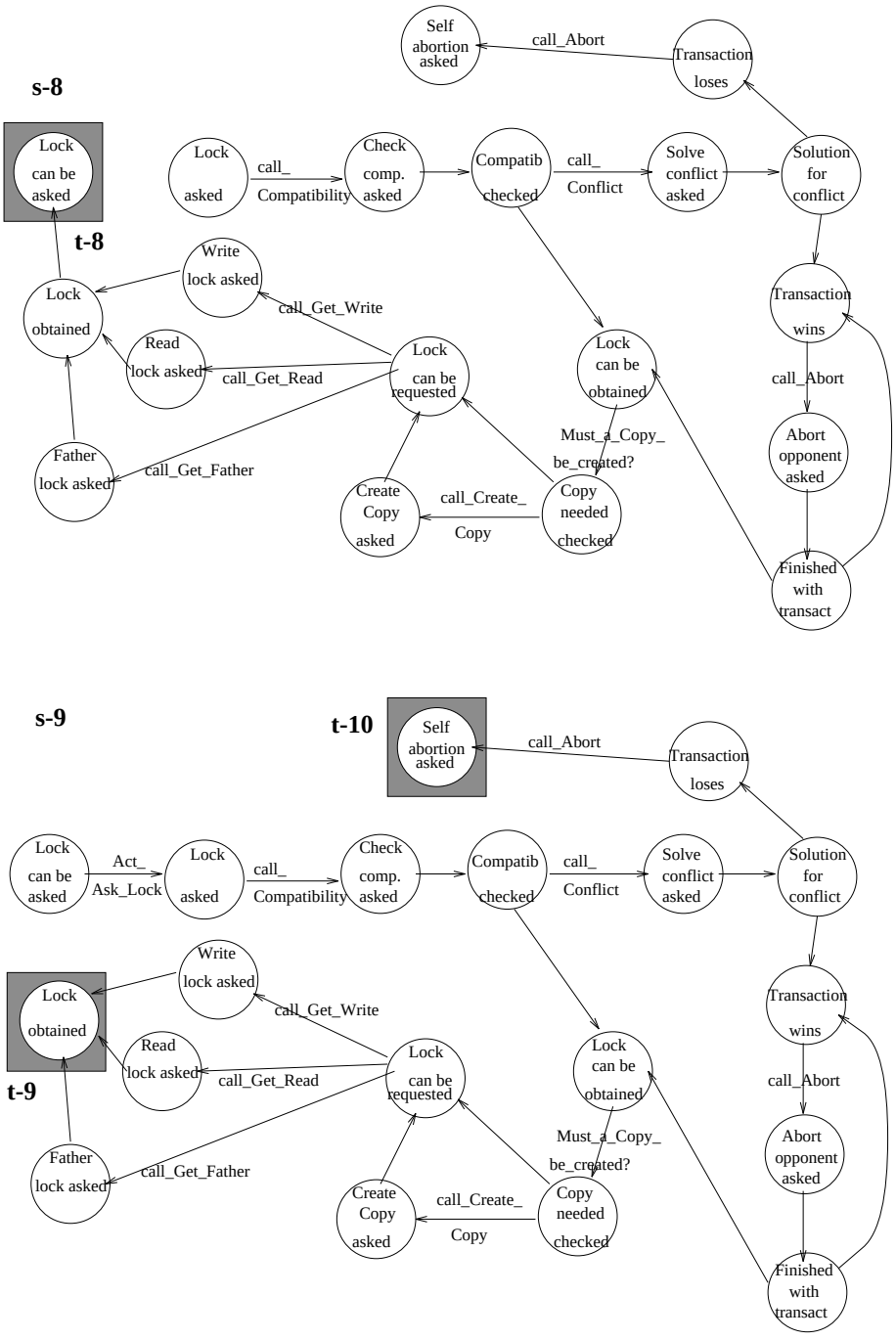
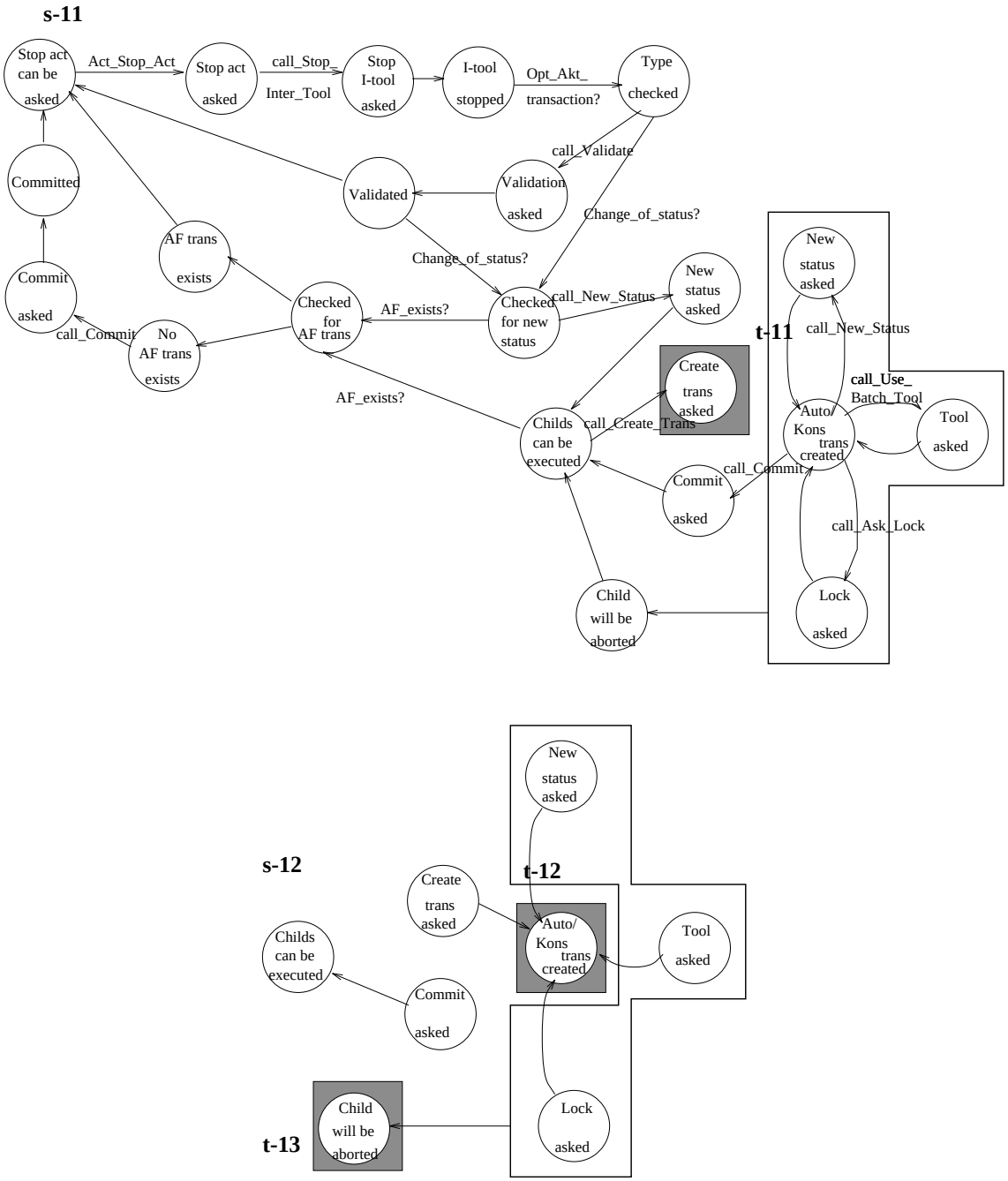
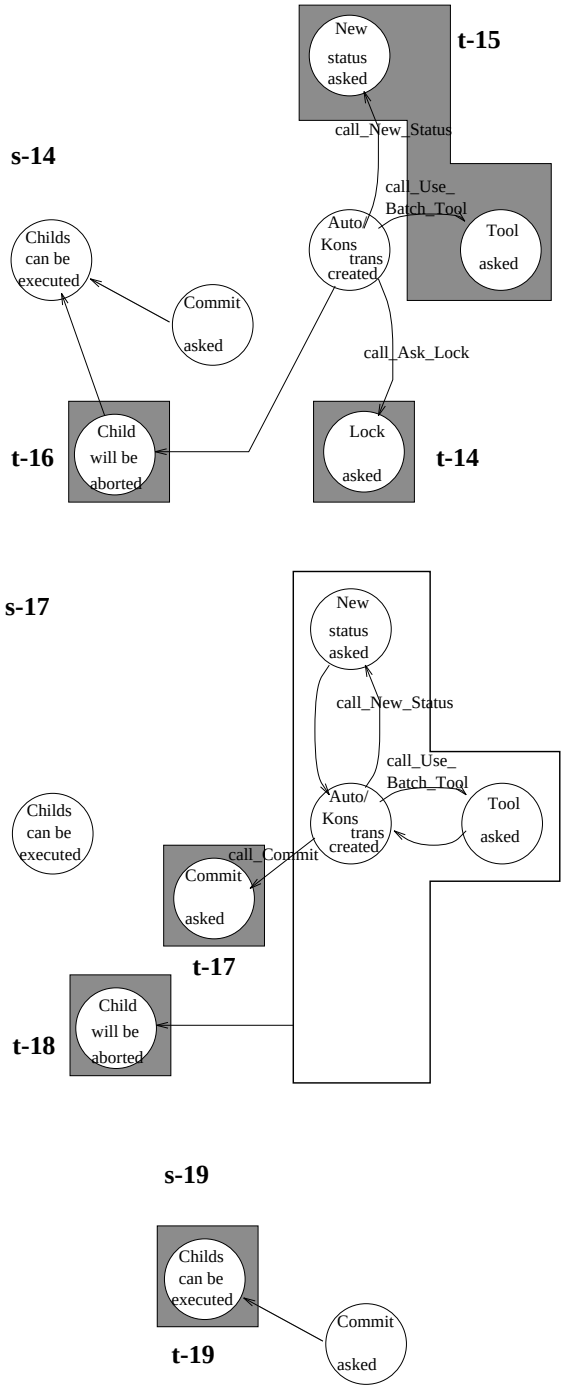


FIGURE 57.int-Ask_Lock's subprocesses and traps w.r.t. Auto Transaction.


s-11
Stop act can be asked
Act_Stop_Act
Stop act asked
call_Stop_ Inter_Tool
Stop I-tool asked
I-tool stopped
Opt_Akt_ transaction?
Type checked
call_Validate
Validation asked
Validated
Committed
Change_of_status?
AF trans exists
Change_of_status?
Commit asked
Checked for AF trans
AF_exists?
Checked for new status
call_New_Status
New status asked
call_Commit
No AF trans exists
AF_exists?
Create trans asked
Childs can be executed
call_Create_Trans
t-11
New status asked
call_New_Status
call_Use_ Batch_Tool
Auto/ Kons trans created
Tool asked
call_Commit
Commit asked
call_Ask_Lock
Child will be aborted
Lock asked

New status asked
s-12
t-12
Create trans asked
Auto/ Kons trans created
Tool asked
Childs can be executed
Commit asked
Lock asked
t-13
Child will be aborted


New
status
asked
t-15
call_New_Status
s-14
call_Use_
Batch_Tool
Childs
can be
executed
Auto/
Kons
trans
created
Tool
asked
Commit
asked
call_Ask_Lock
Child
will be
aborted
Lock
asked
t-16
t-14
s-17
New
status
asked
call_New_Status
call_Use_
Batch_Tool
Childs
can be
executed
Auto/
Kons
trans
created
Tool
asked
call_Commit
Commit
asked
t-17
Child
will be
aborted
t-18
s-19
Childs
can be
executed
Commit
asked
t-19

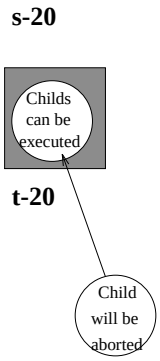


FIGURE 58.int-Stop_Act's subprocesses and traps w.r.t. Auto Transaction.

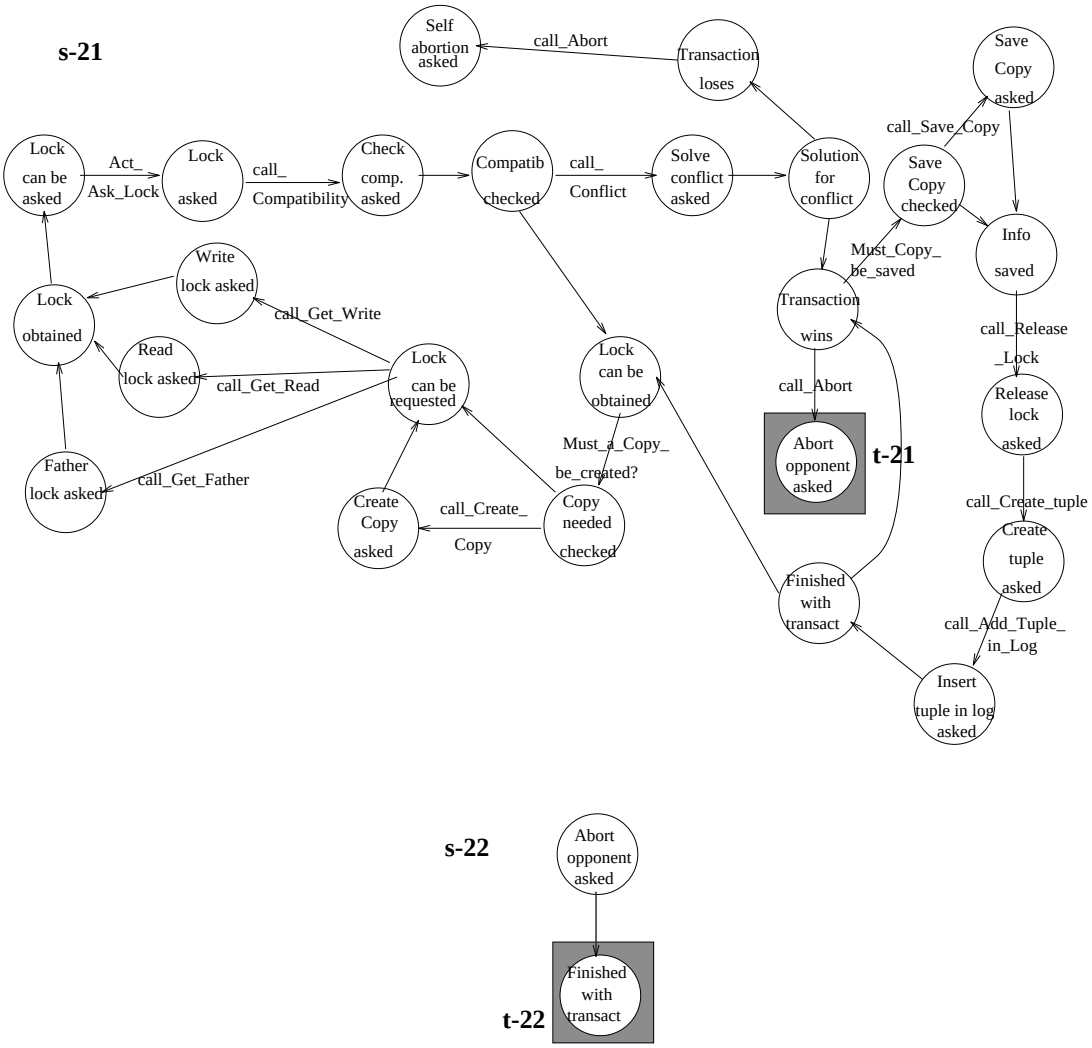


FIGURE 59.int-Ask_Lock's subprocesses and traps from Kons -, Pess_Akt - or Pess_AF Transaction w.r.t. Auto Transaction.

The above employee has to be an instance of the class Kons -, Pess_Akt or Pess_AF Transaction, because only such a transaction can win a conflict from an Auto Transaction.

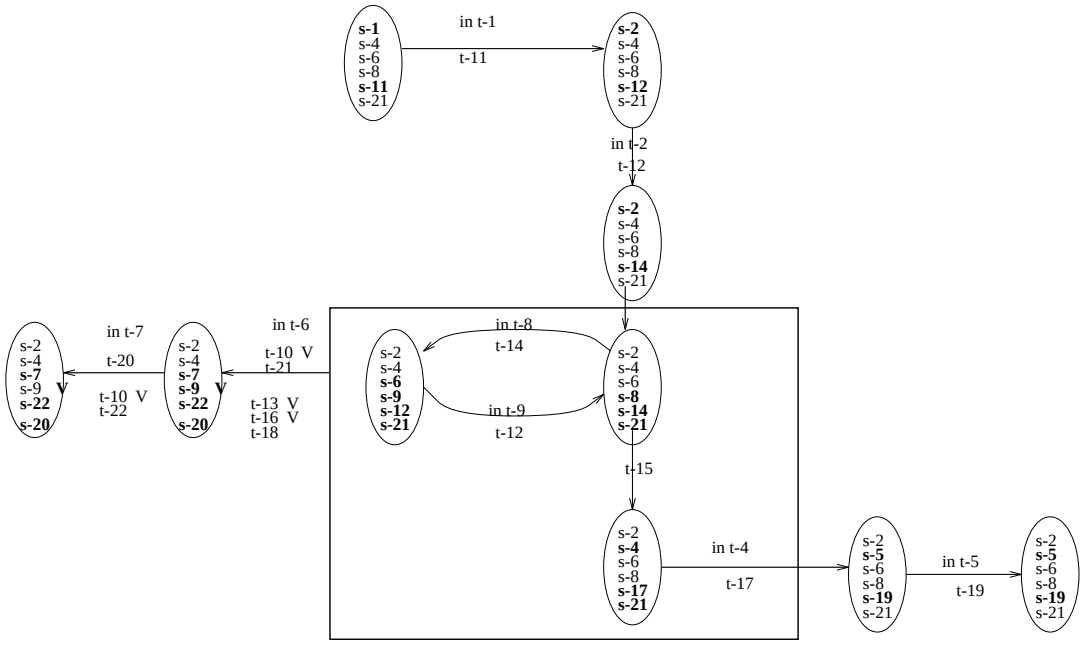


FIGURE 60. Auto Transaction, manager of six employees.

The next manager is Pess_Akt Transaction. Its extra employees are:

Ask_Lock (same instance)

Start_Act

Stop_Act

Ask_Lock (other instance)

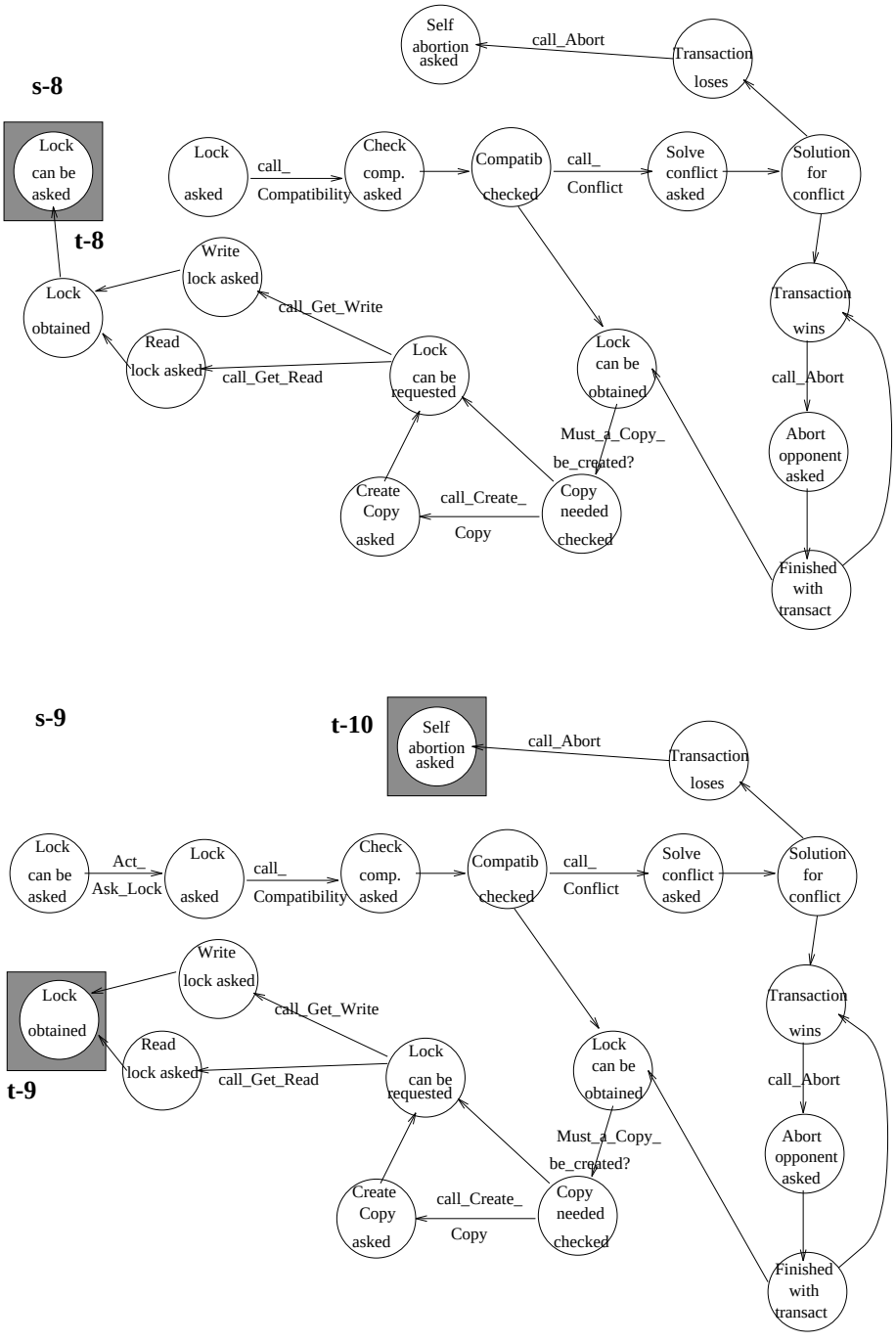
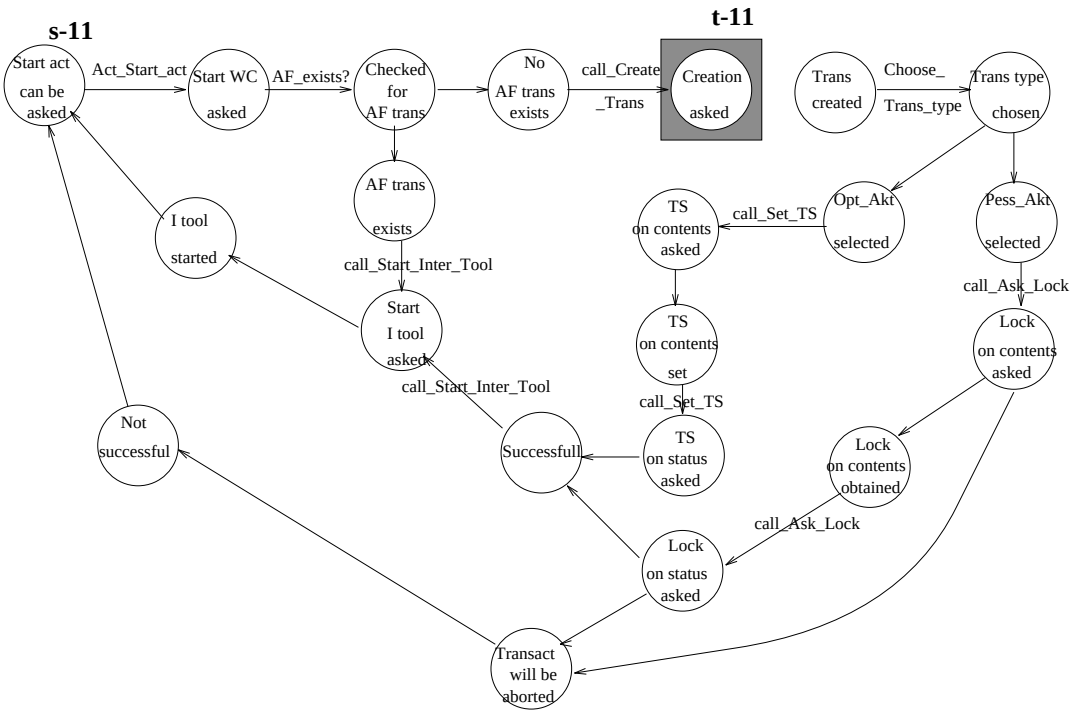
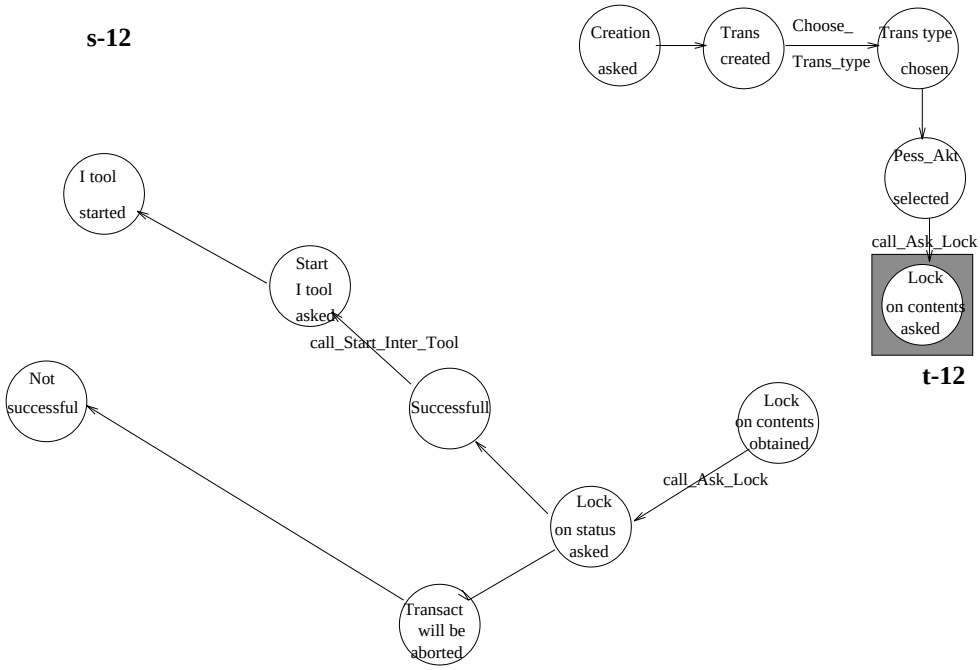


FIGURE 61.int-Ask_Lock's subprocesses and traps w.r.t. Pess_Akt Transaction.


s-11
t-11
Start act
can be
asked
Act_Start_act
Start WC
asked
AF_exists?
Checked
for
AF trans
No
AF trans
exists
call_Create
_Trans
Creation
asked
Trans
created
Choose_
Trans_type
Trans type
chosen
AF trans
exists
I tool
started
call_Start_Inter_Tool
Start
I tool
asked
call_Start_Inter_Tool
Not
successful
Successfull
TS
on contents
asked
call_Set_TS
Opt_Akt
selected
Pess_Akt
selected
call_Ask_Lock
TS
on contents
set
call_Set_TS
TS
on status
asked
Lock
on contents
asked
Lock
on contents
obtained
call_Ask_Lock
Lock
on status
asked
Transact
will be
aborted


s-12
Creation
asked
Trans
created
Choose_
Trans_type
Trans type
chosen
Pess_Akt
selected
call_Ask_Lock
Lock
on contents
asked
t-12
I tool
started
Start
I tool
asked
call_Start_Inter_Tool
Not
successful
Successfull
Lock
on contents
obtained
call_Ask_Lock
Lock
on status
asked
Transact
will be
aborted

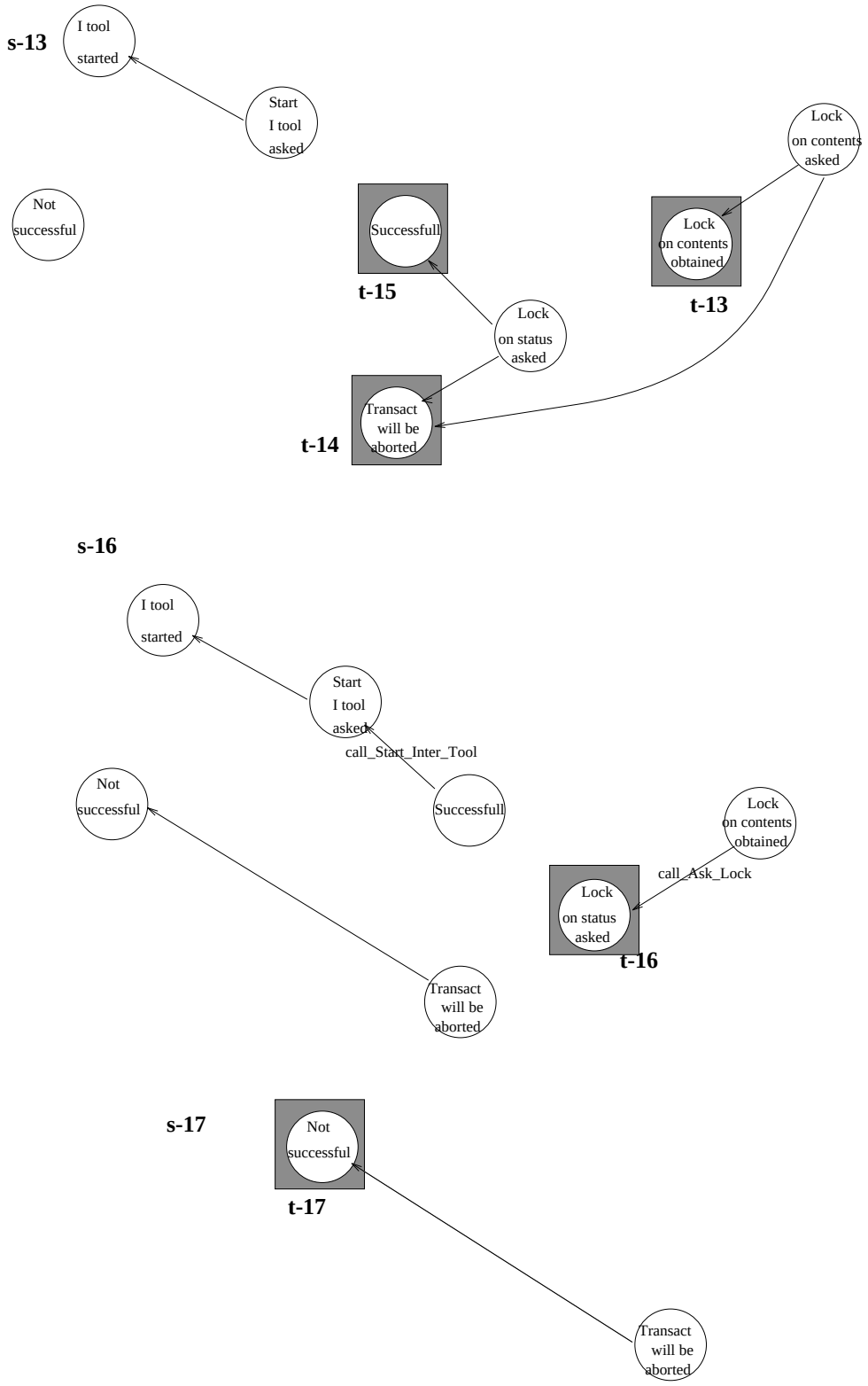


FIGURE 62.int-Start_Act's subprocesses and traps w.r.t. Pess_Akt Transaction.

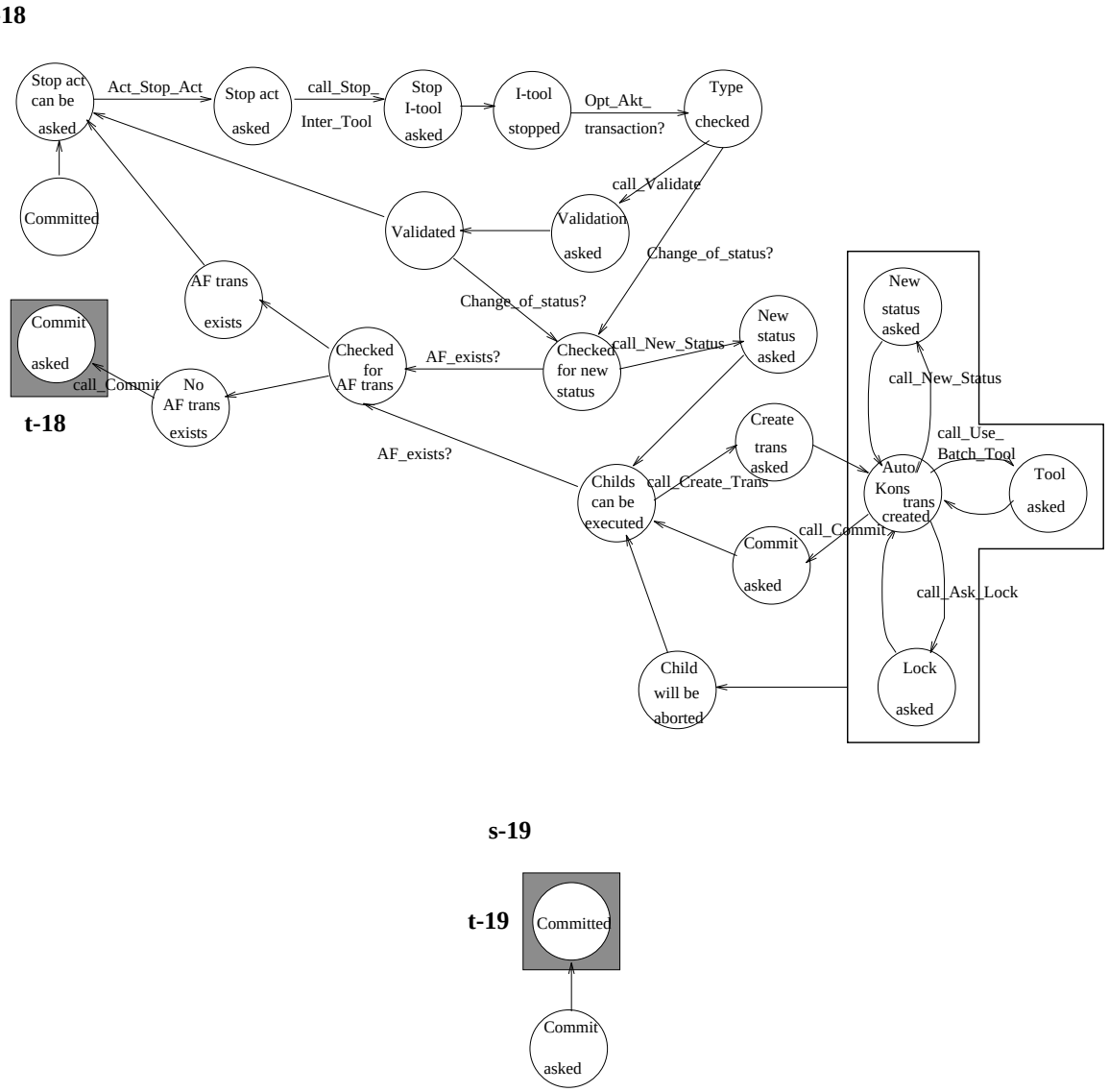


FIGURE 63.int-Stop_Act's subprocesses and traps w.r.t. Pess_Akt Transaction.

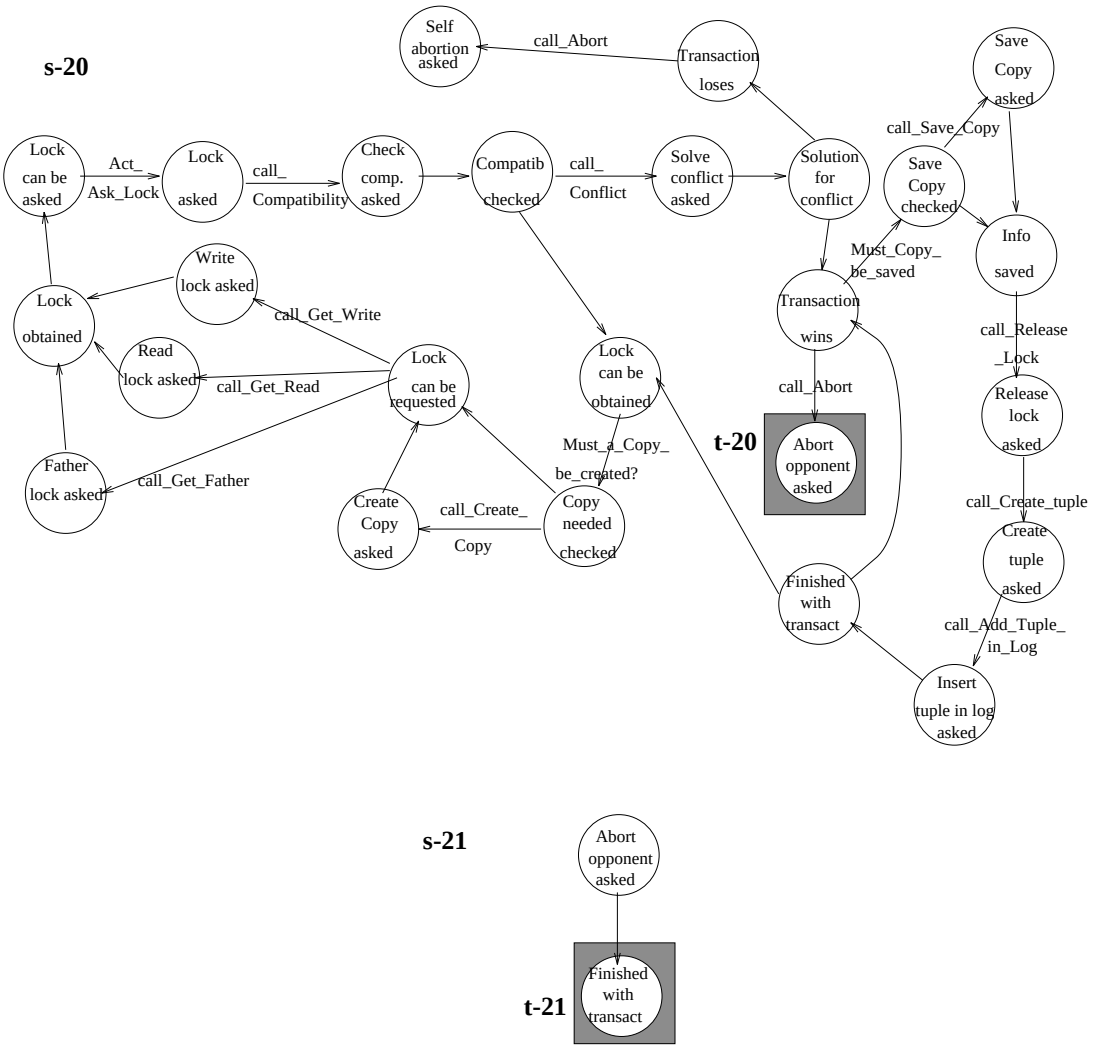


FIGURE 64.int-Ask_Lock's subprocesses and traps from Kons Transaction w.r.t. Pess_Akt Transaction.

The above employee has to be an instance of the class Kons Transaction, because only a Kons transaction can win a conflict from a Pess_Akt Transaction.

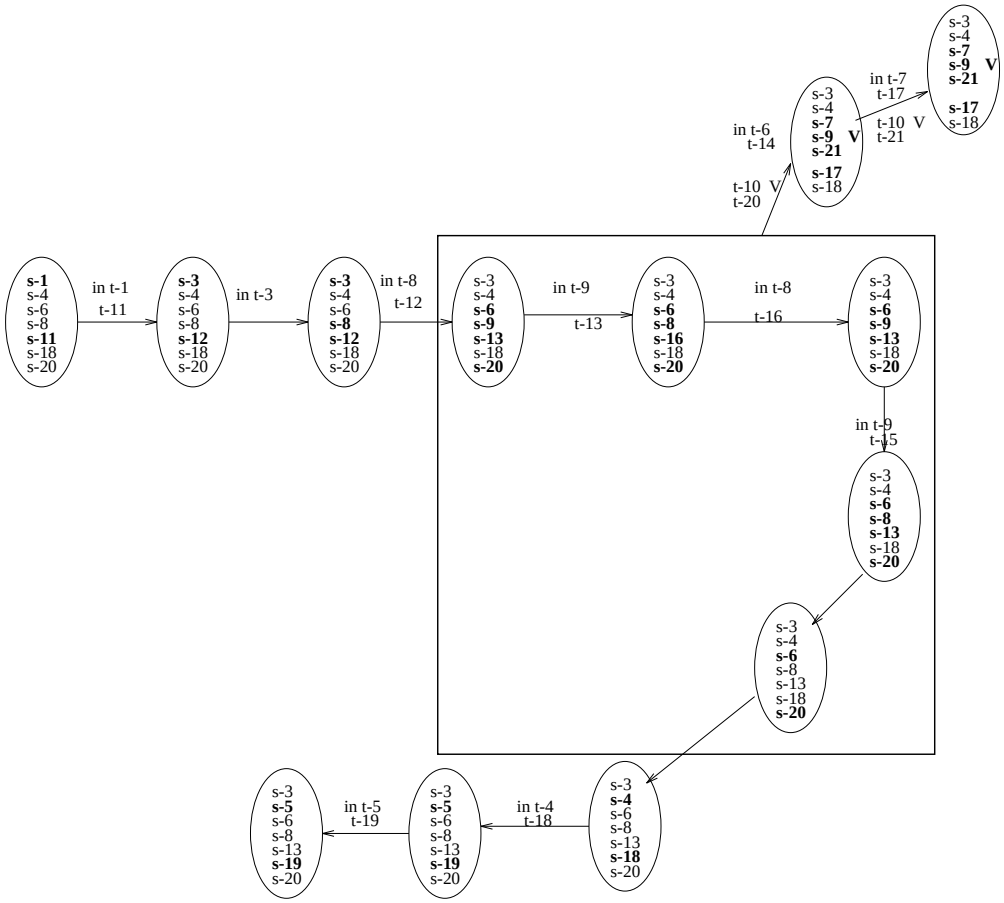


FIGURE 65. Pess_Akt Transaction, manager of seven employees.

The next manager is Pess_AF Transaction. Its extra employees are:

Ask_Lock (same instance)
Remove_Lock
Update_Lock
Refresh
Start_WC
Stop_WC
Ask_Lock (other instance)

Ask_Lock (same instance), see Pess_Akt manager

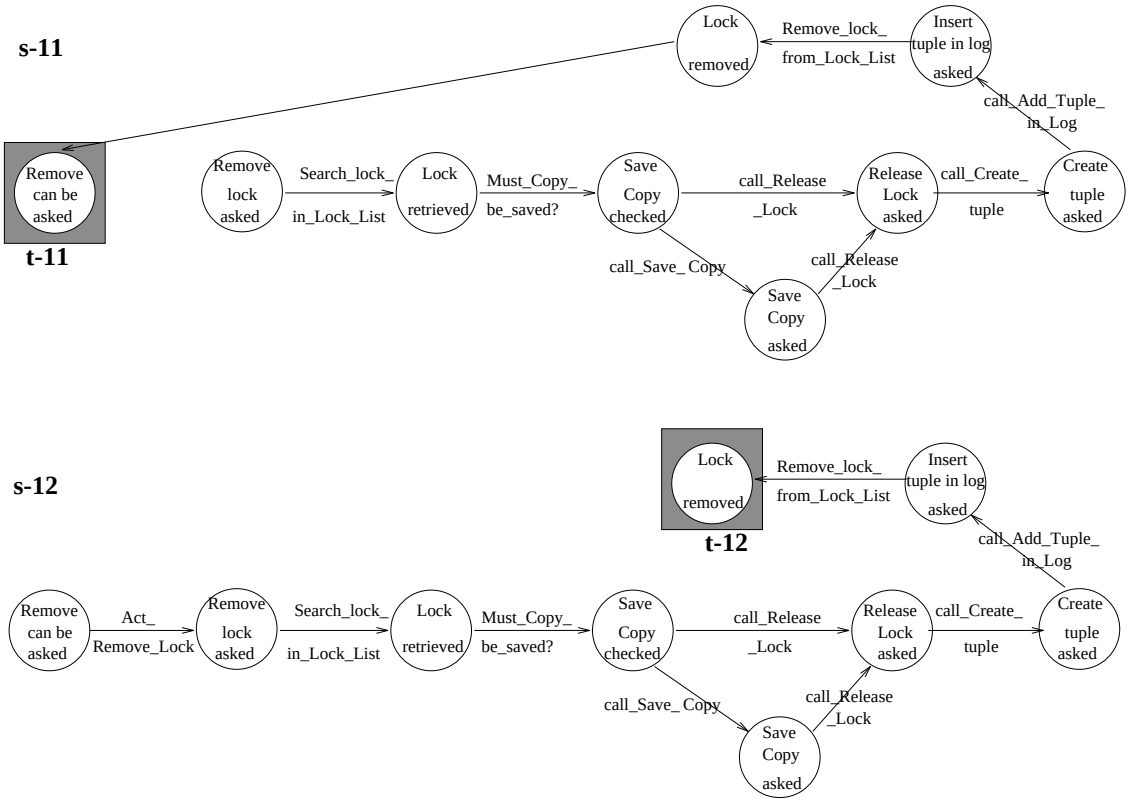
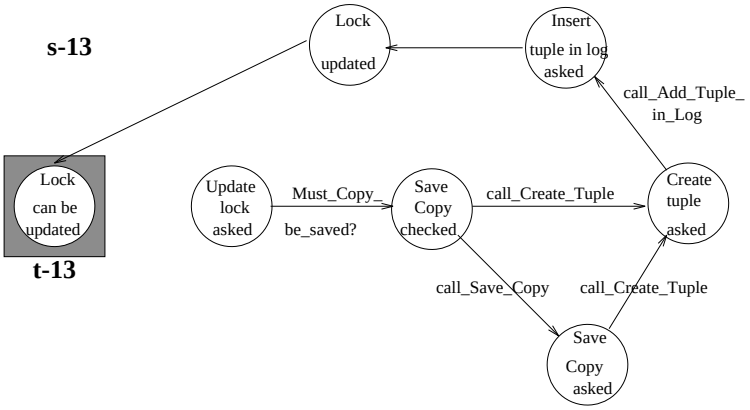


FIGURE 66.int-Remove_Lock's subprocesses and traps w.r.t. Pess_AF Transaction.



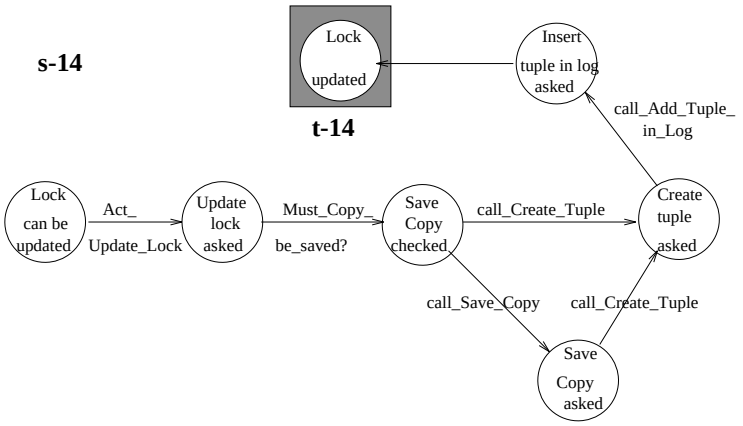
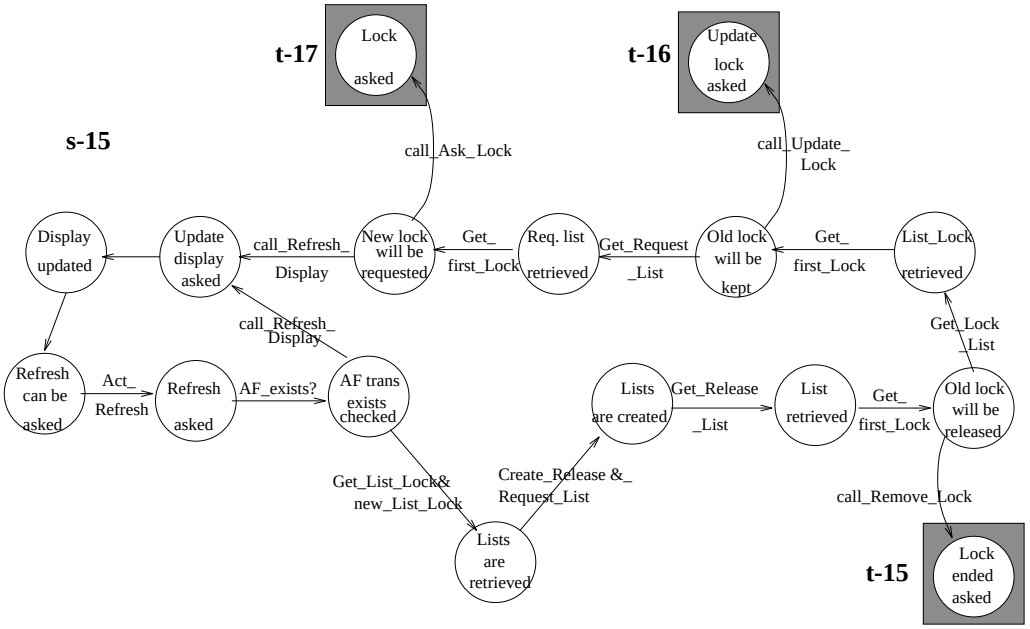


FIGURE 67.int-Update_Lock's subprocesses and traps w.r.t. Pess_AF Transaction.



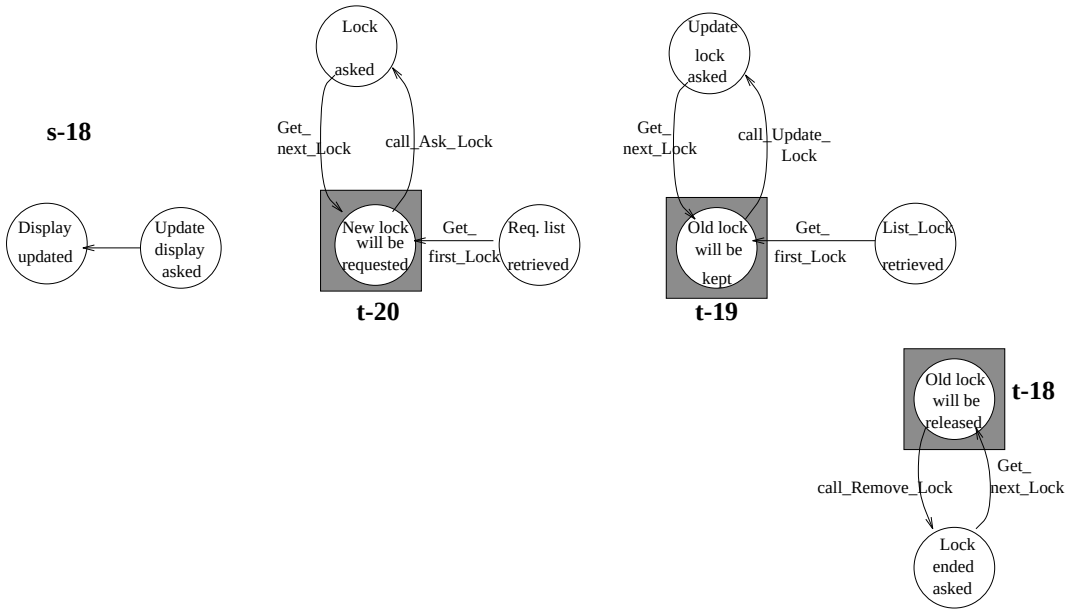
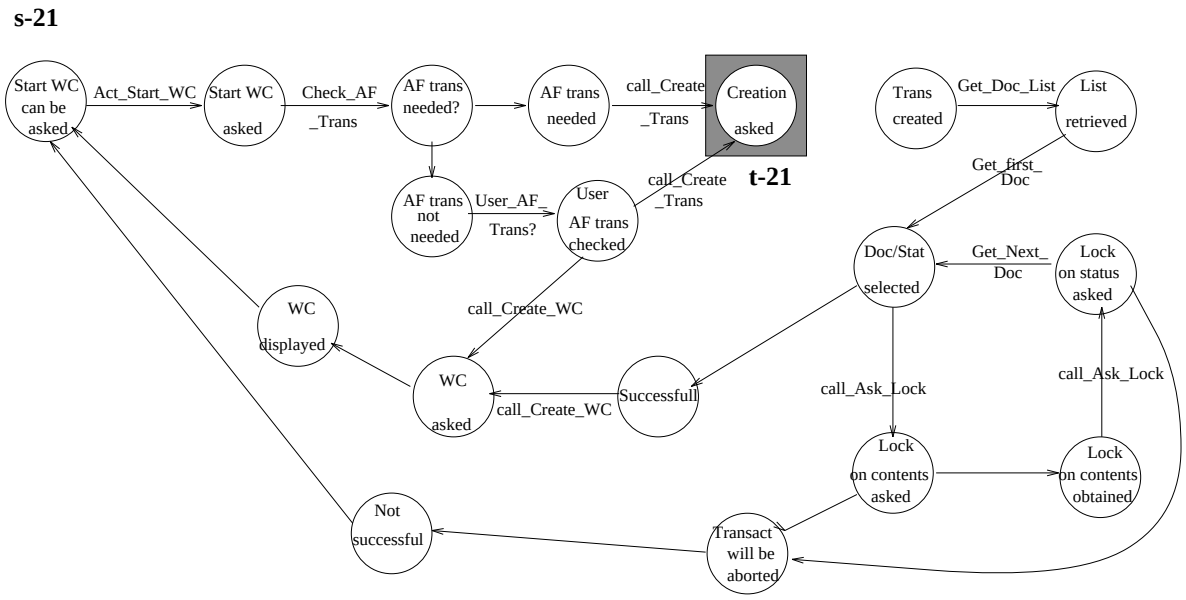


FIGURE 68.int-Refresh's subprocesses and traps w.r.t. Pess_AF Transaction.



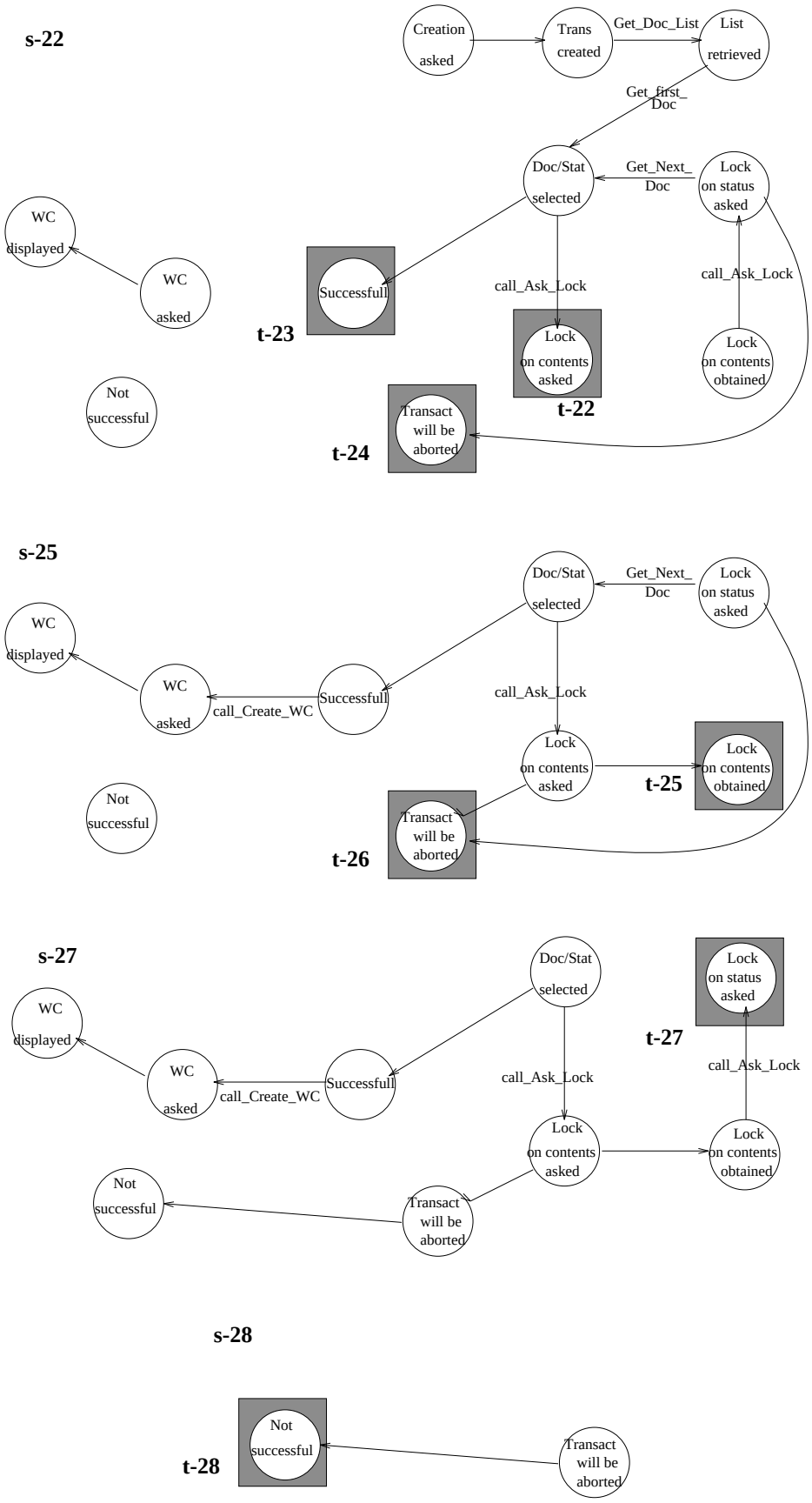


FIGURE 68.int-Start_WC's subprocesses and traps w.r.t. Pess_AF Transaction.

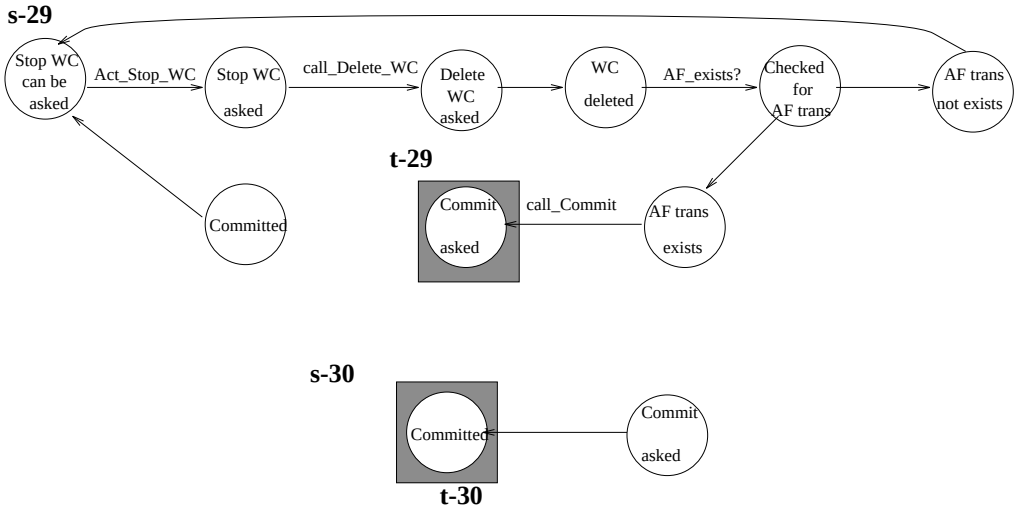


FIGURE 69.int-Stop_WC's subprocesses and traps w.r.t. Pess_AF Transaction.

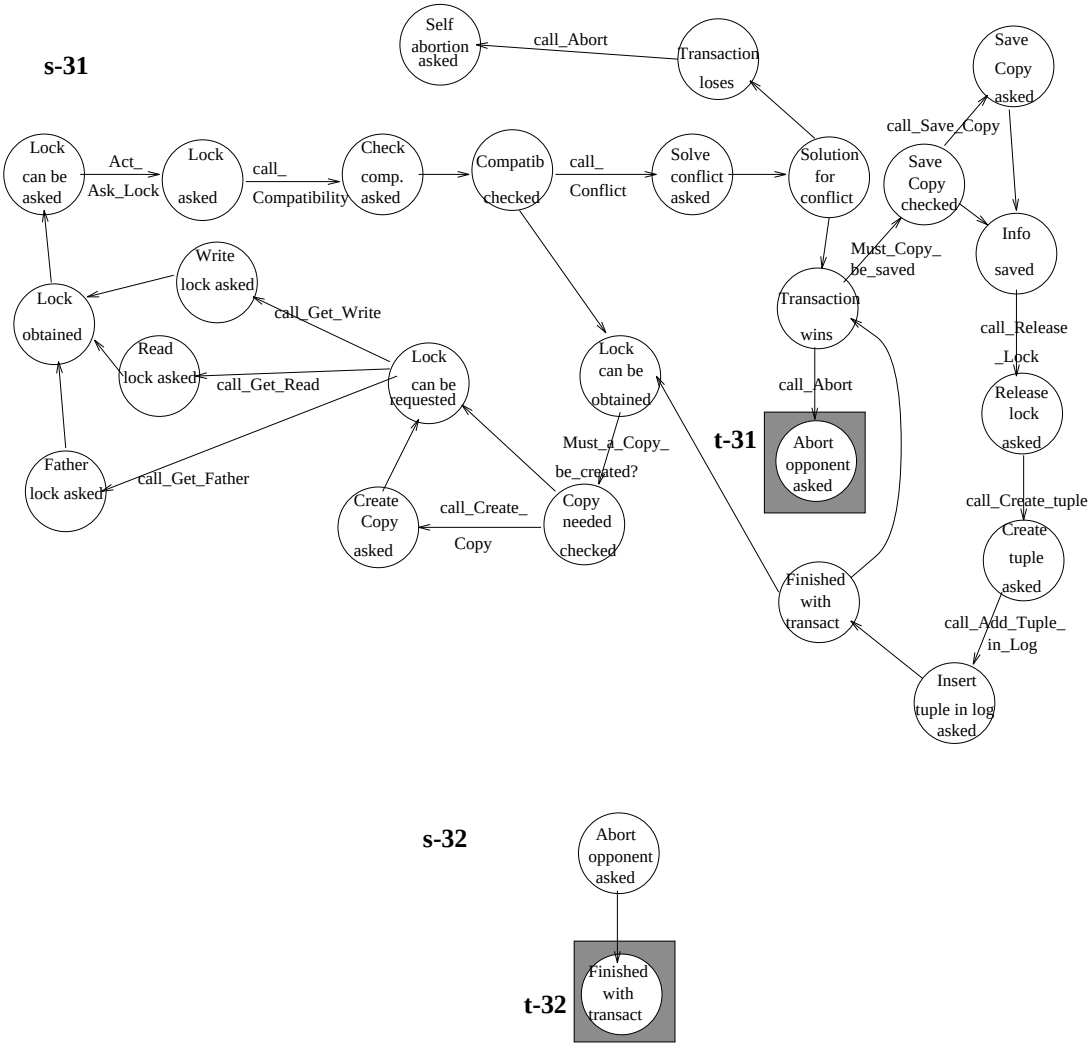


FIGURE 70.int-Ask_Lock's subprocesses and traps from Kons Transaction w.r.t. Pess_AF Transaction.

The above employee has to be an instance of the class Kons Transaction, because only a Kons transaction can win a conflict from a Pess_AF Transaction.

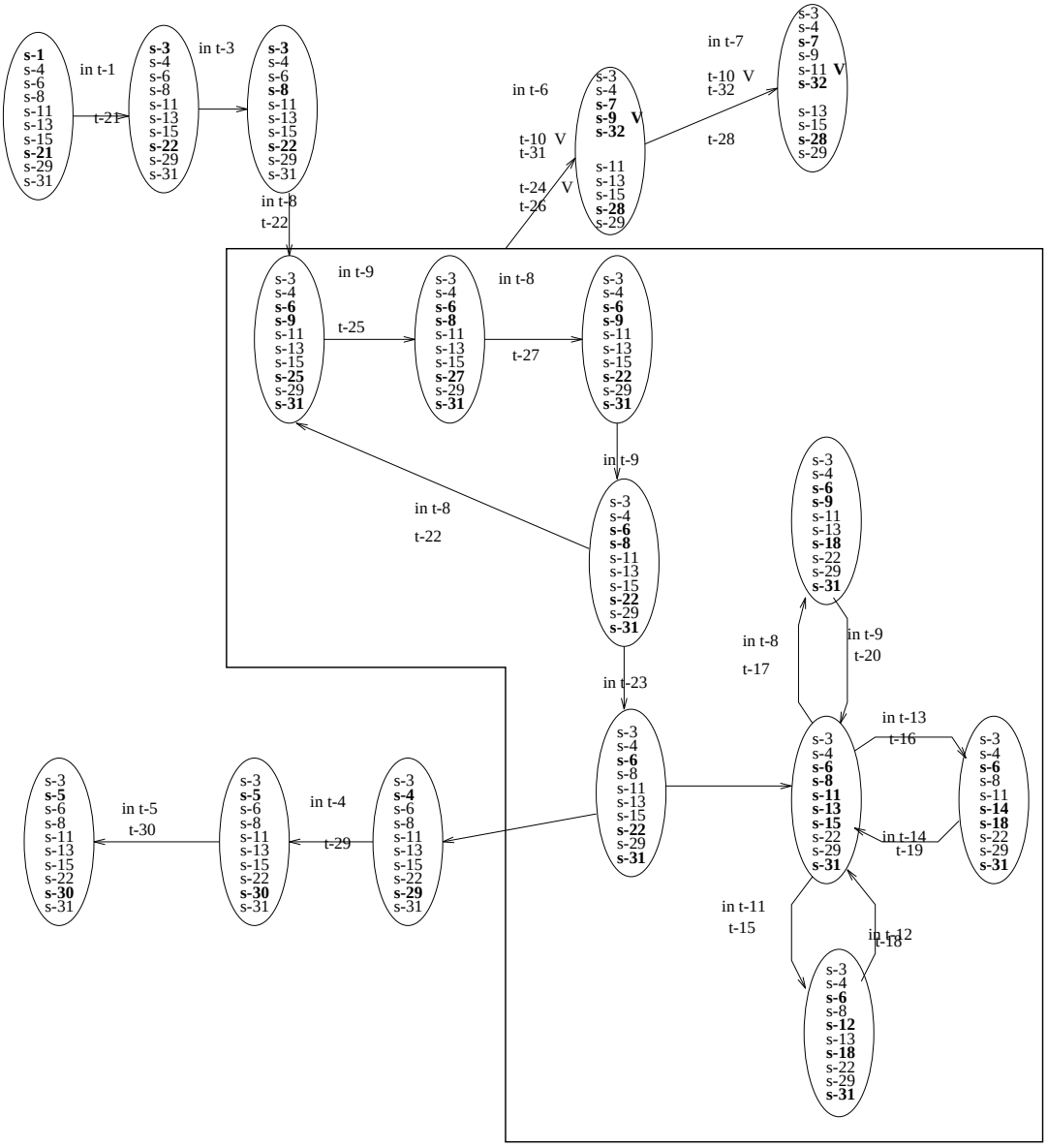


FIGURE 71. Pess_AF Transaction, manager of seven employees.

The next manager is Opt_Akt Transaction. Its extra employees are:

Set_TS
Validate
Start_Act
Stop_Act

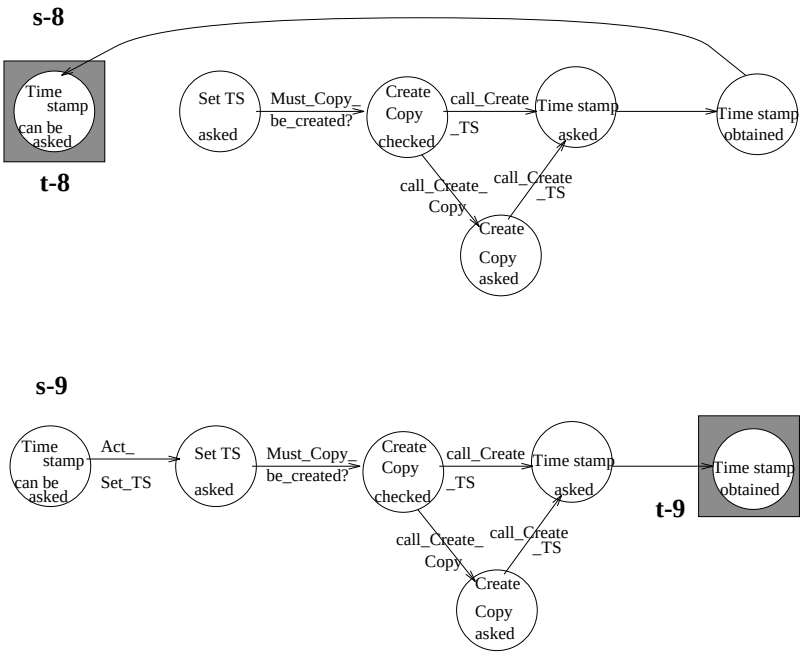
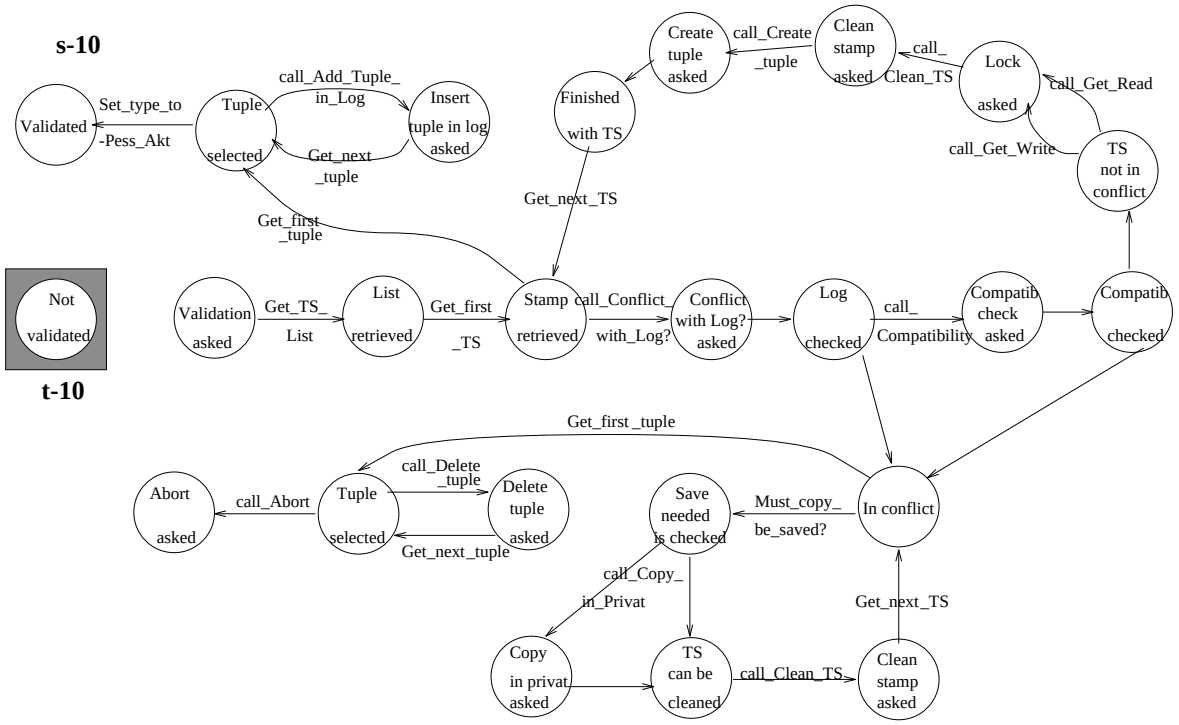


FIGURE 72.int-Set_TS's subprocesses and traps w.r.t. Opt_Akt Transaction.



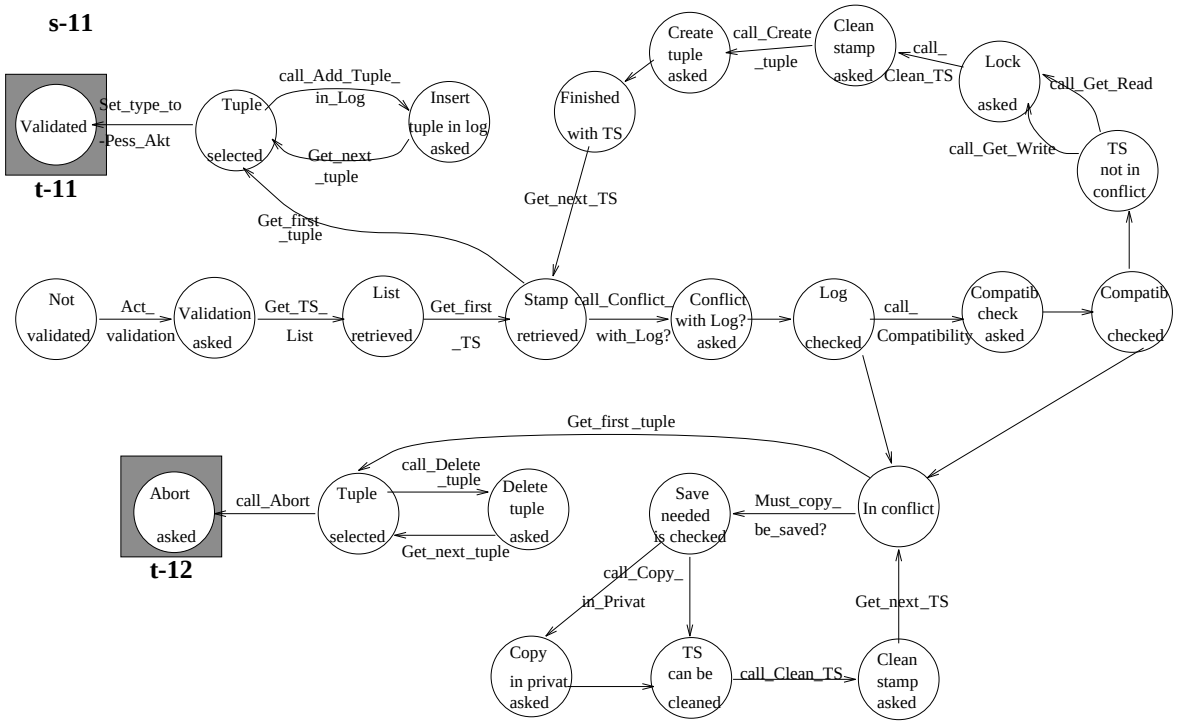
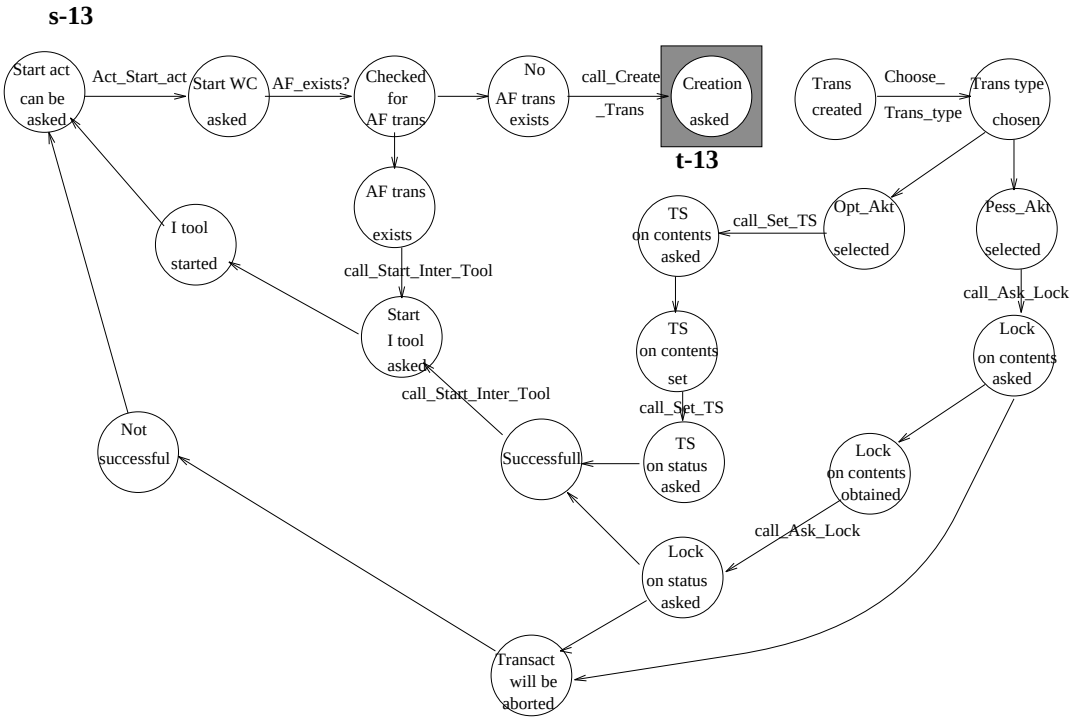


FIGURE 73.int-Validate's subprocesses and traps w.r.t. Opt_Akt Transaction.



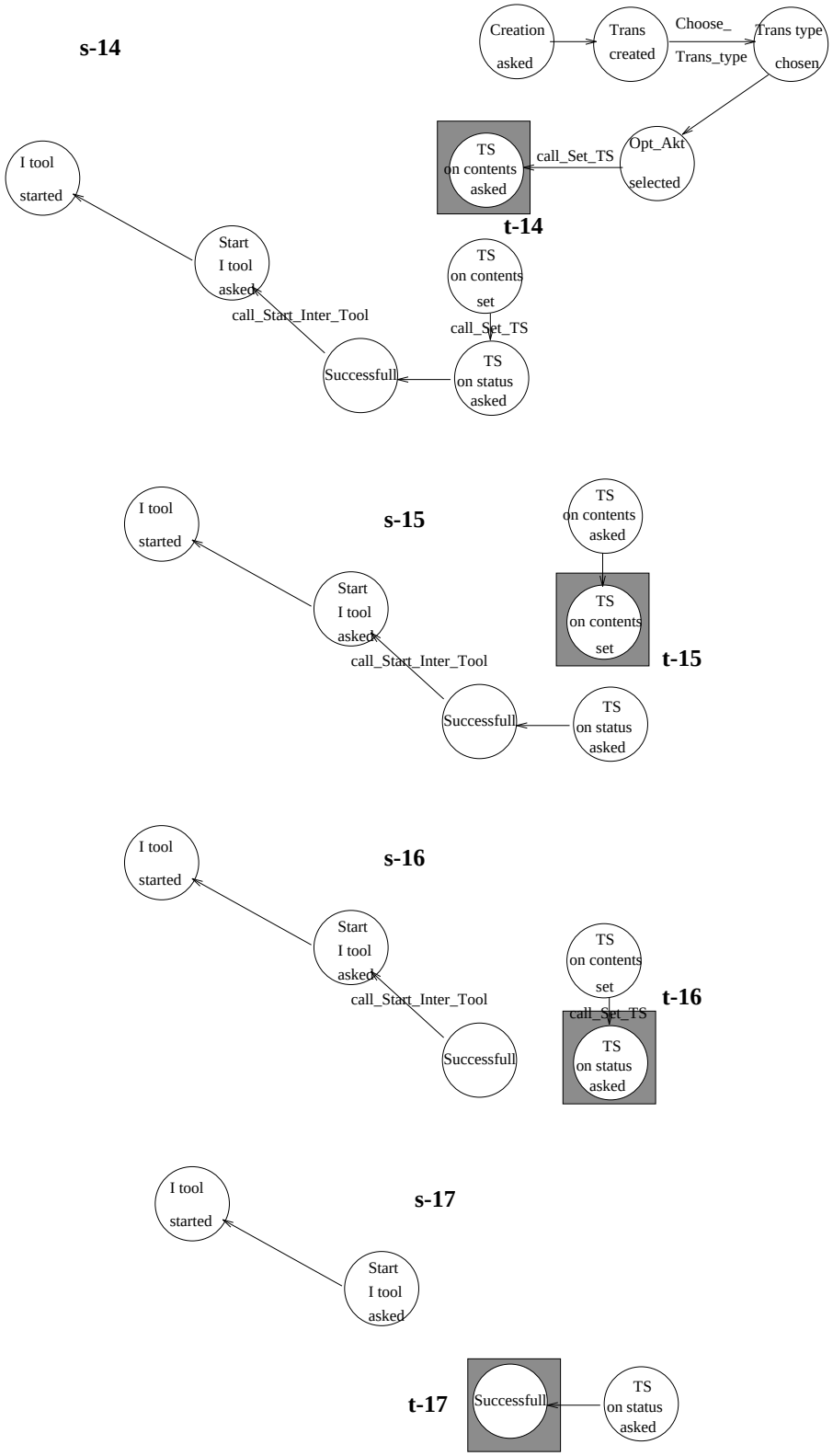
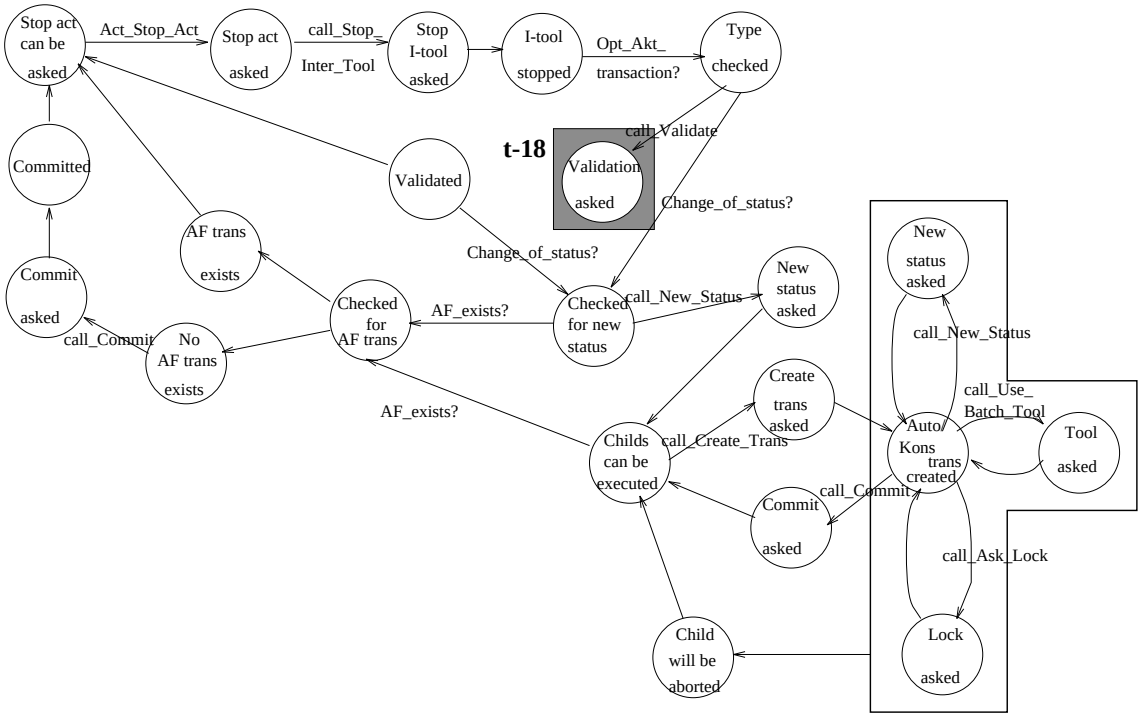
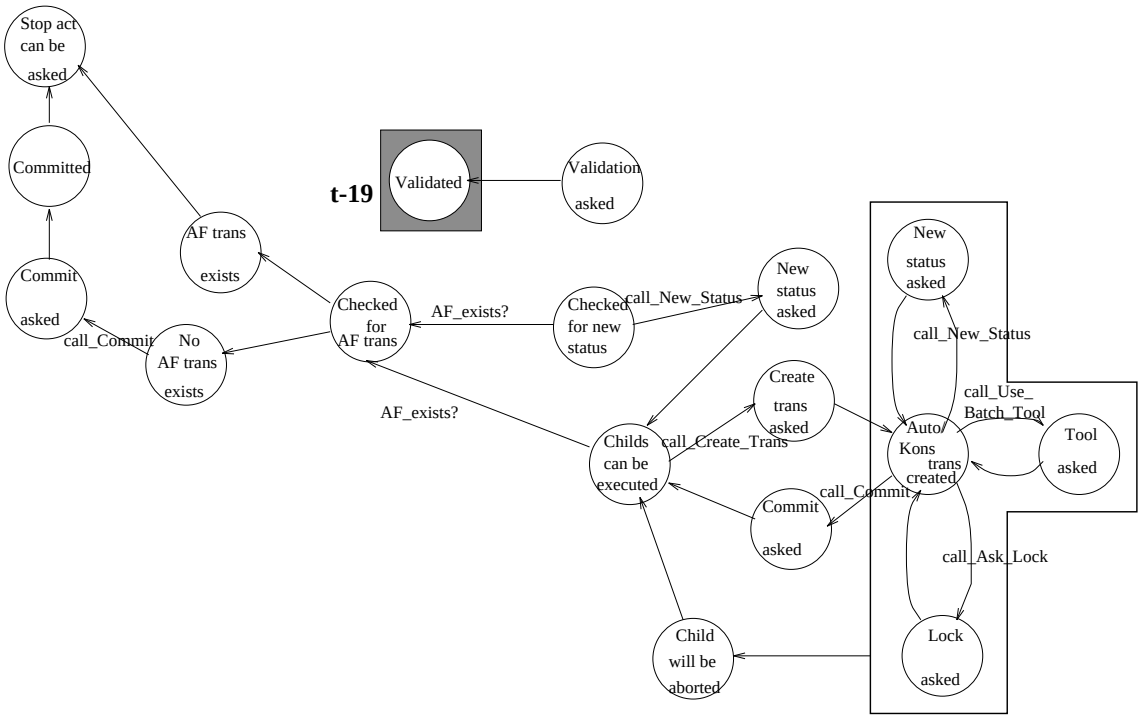


FIGURE 74.int-Start_Act's subprocesses and traps w.r.t. Opt_Akt Transaction.

s-18


Stop act can be asked
Act_Stop_Act
Stop act asked
call_Stop_ Inter_Tool
Stop I-tool asked
I-tool stopped
Opt_Akt_ transaction?
Type checked
call_Validate
t-18
Validation asked
Validated
Committed
Change_of_status?
AF trans exists
Change_of_status?
Commit asked
Checked for AF trans
AF_exists?
Checked for new status
call_New_Status
New status asked
call_Commit
No AF trans exists
AF_exists?
Create trans asked
call_Use_ Batch_Tool
Childs can be executed
call_Create_Trans
Auto Kons trans created
Tool asked
call_Commit
Commit asked
call_New_Status
call_Ask_Lock
Child will be aborted
Lock asked

s-19


Stop act can be asked
Committed
t-19
Validated
Validation asked
AF trans exists
Commit asked
Checked for AF trans
AF_exists?
Checked for new status
call_New_Status
New status asked
call_Commit
No AF trans exists
AF_exists?
Create trans asked
call_Use_ Batch_Tool
Childs can be executed
call_Create_Trans
Auto Kons trans created
Tool asked
call_Commit
Commit asked
call_New_Status
call_Ask_Lock
Child will be aborted
Lock asked

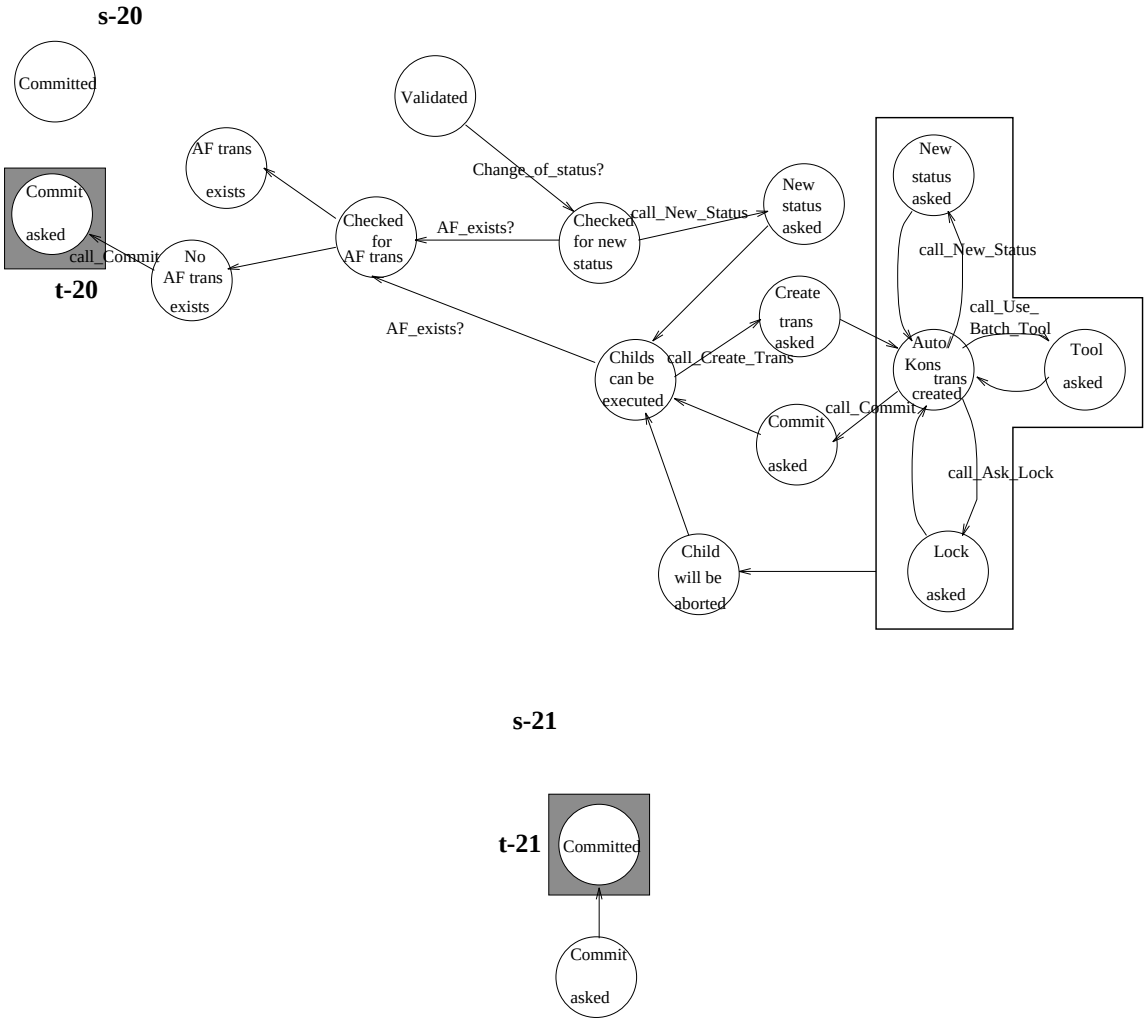


FIGURE 75.int-Stop_Act's subprocesses and traps w.r.t. Opt_Akt Transaction.

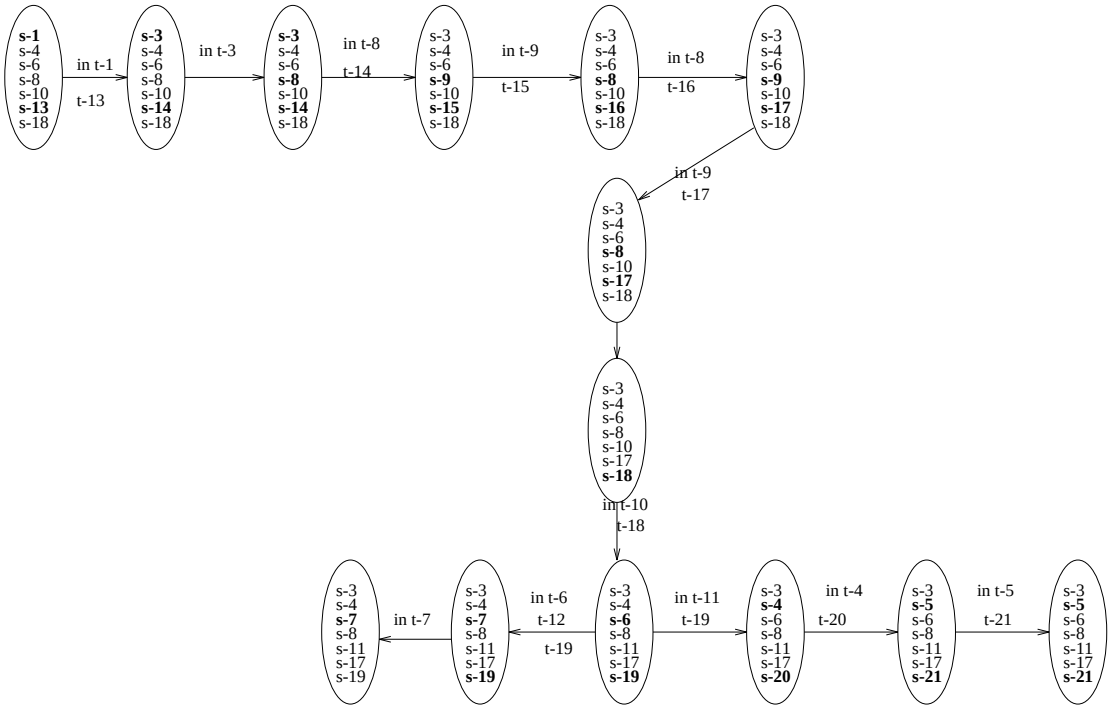


FIGURE 76. Opt_Akt Transaction, manager of seven employees.

The next manager is Working Context. Its employees are:

Create_WC
Delete_WC
Refresh_WC
Start_WC
Stop_WC
Refresh
Abort

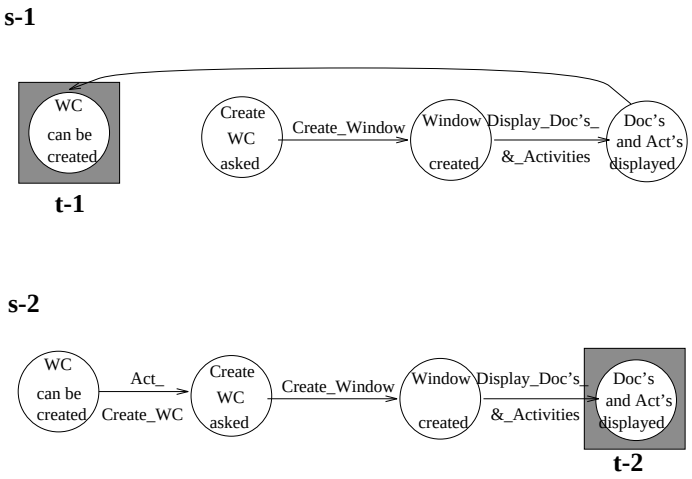


FIGURE 77.int-Create_WC's subprocesses and traps w.r.t. Working Context.

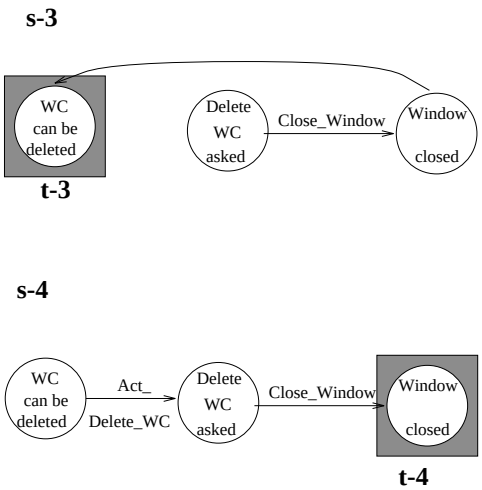


FIGURE 78.int-Delete_WC's subprocesses and traps w.r.t. Working Context.

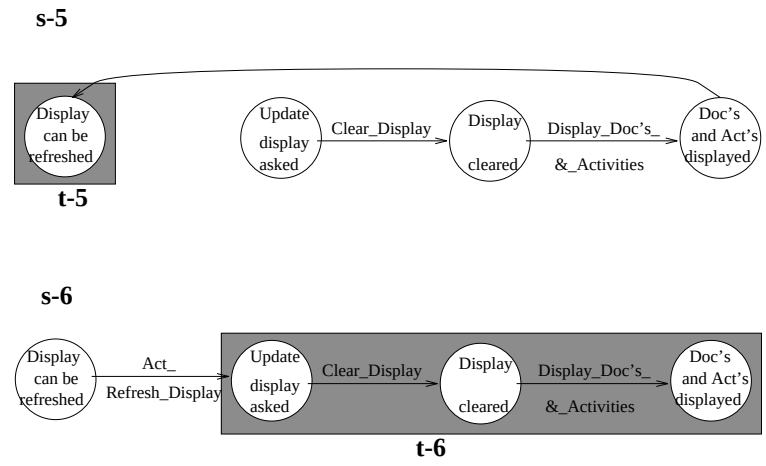


FIGURE 79.int-Refresh_WC's subprocesses and traps w.r.t. Working Context.

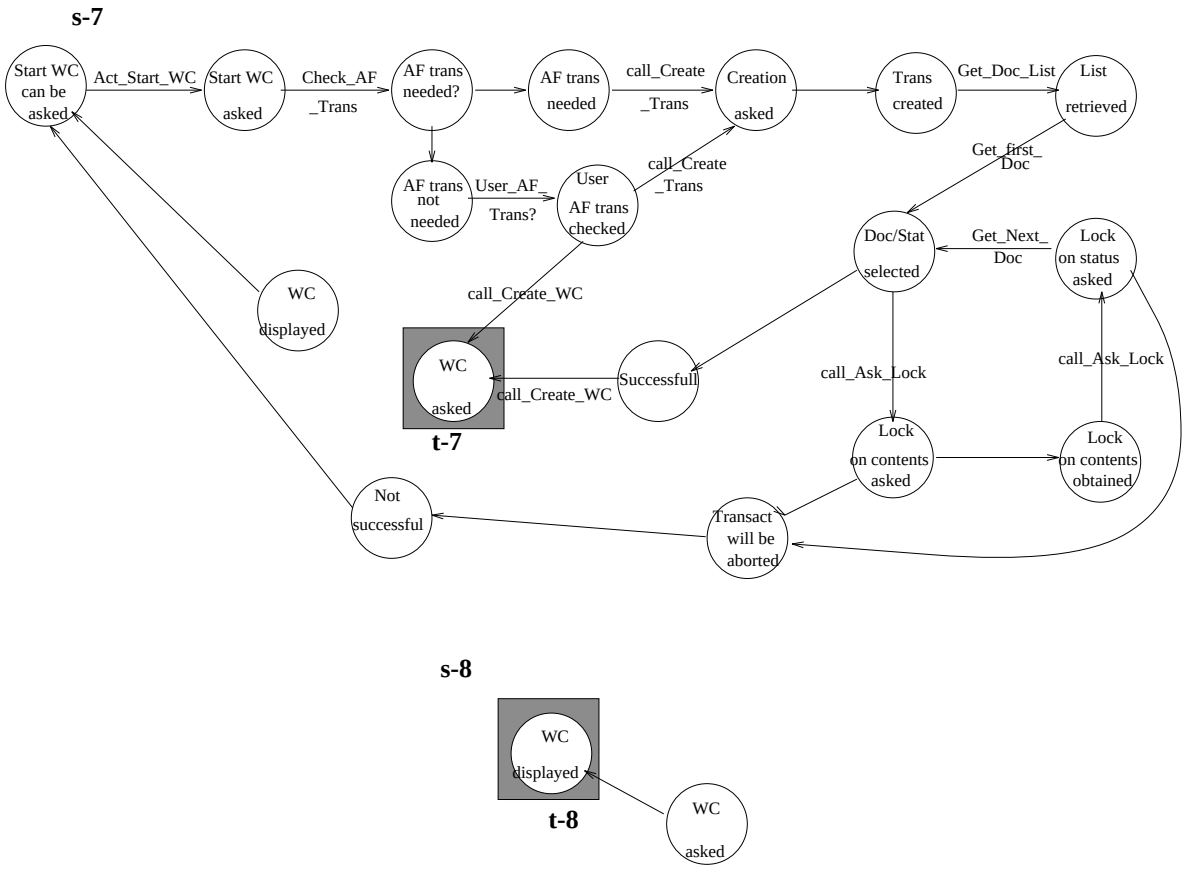


FIGURE 80.int-Start_WC's subprocesses and traps w.r.t. Working Context.

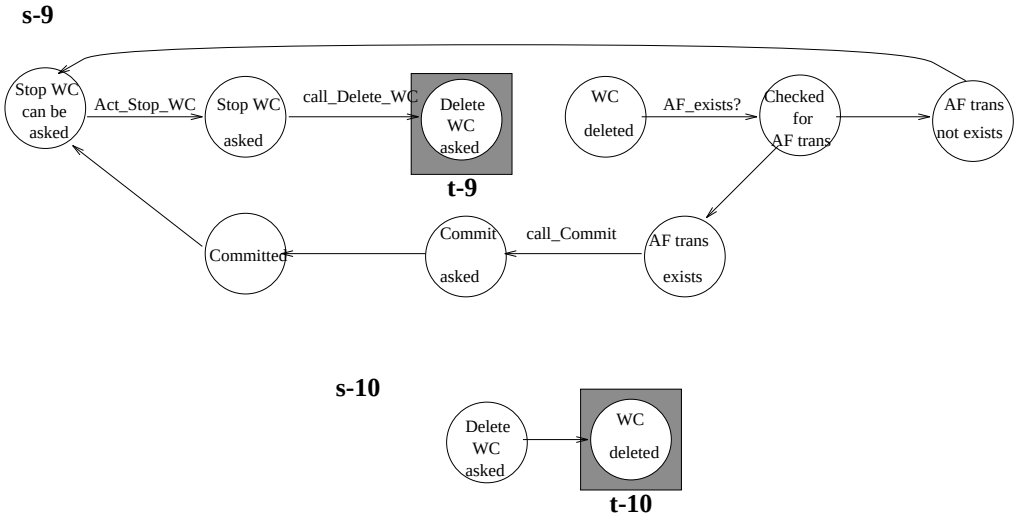


FIGURE 81.int-Stop_WC's subprocesses and traps w.r.t. Working Context.

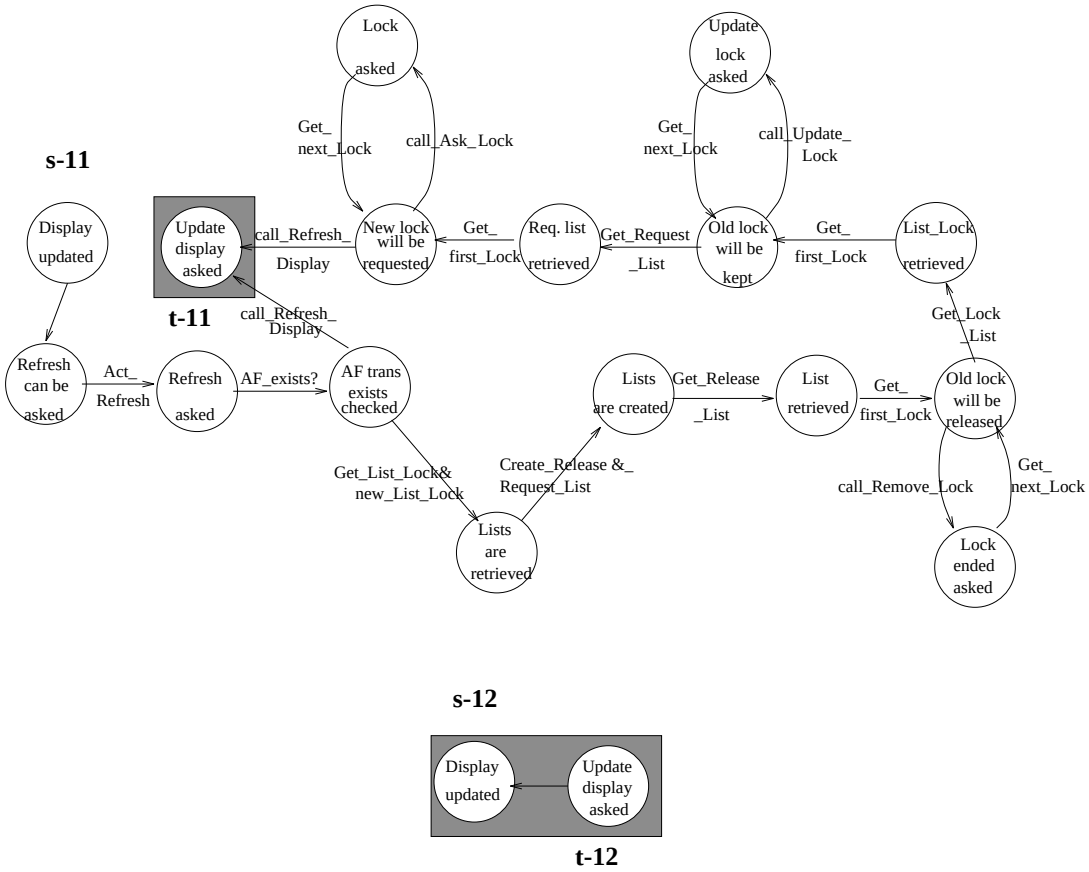


FIGURE 82.int-Refresh's subprocesses and traps w.r.t. Working Context.

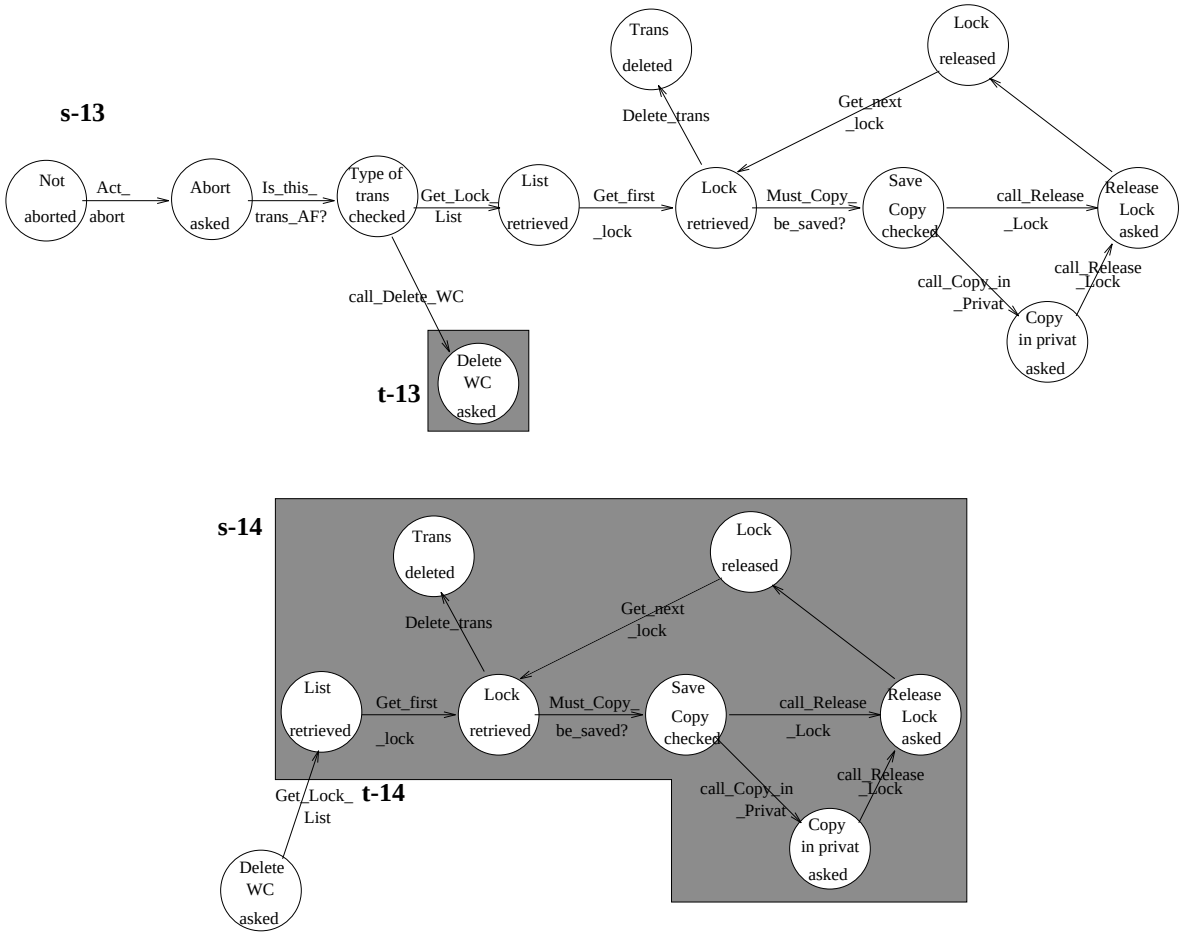


FIGURE 83.int-Abort's subprocesses and traps w.r.t. Working Context.

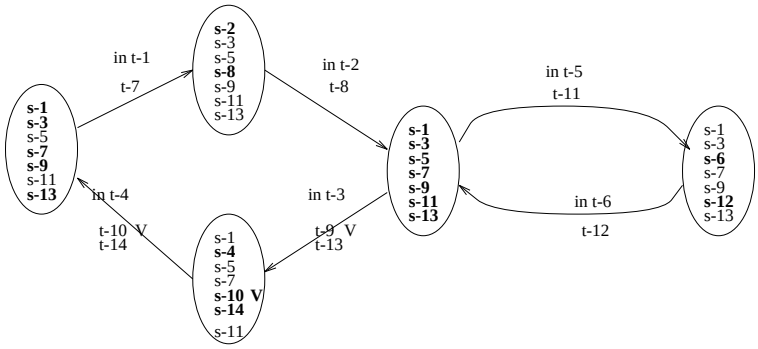


FIGURE 83.Working Context, manager of seven employees.

The next manager is Process Engine. Its employees are:

Start_WC
Stop_WC
Start_Act
Stop_Act
Refresh

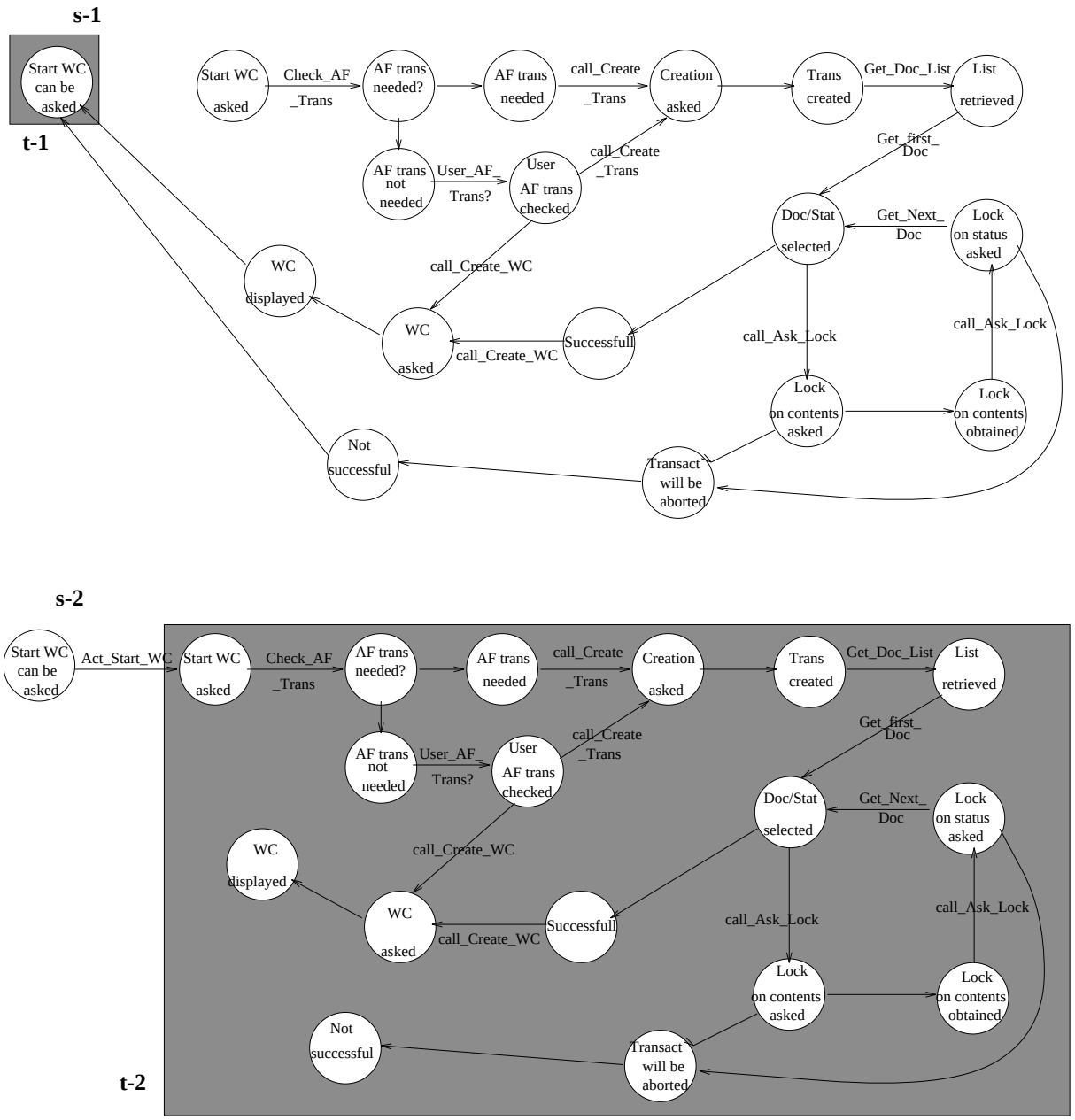


FIGURE 83.int-Start_WC's subprocesses and traps w.r.t. Process Engine.

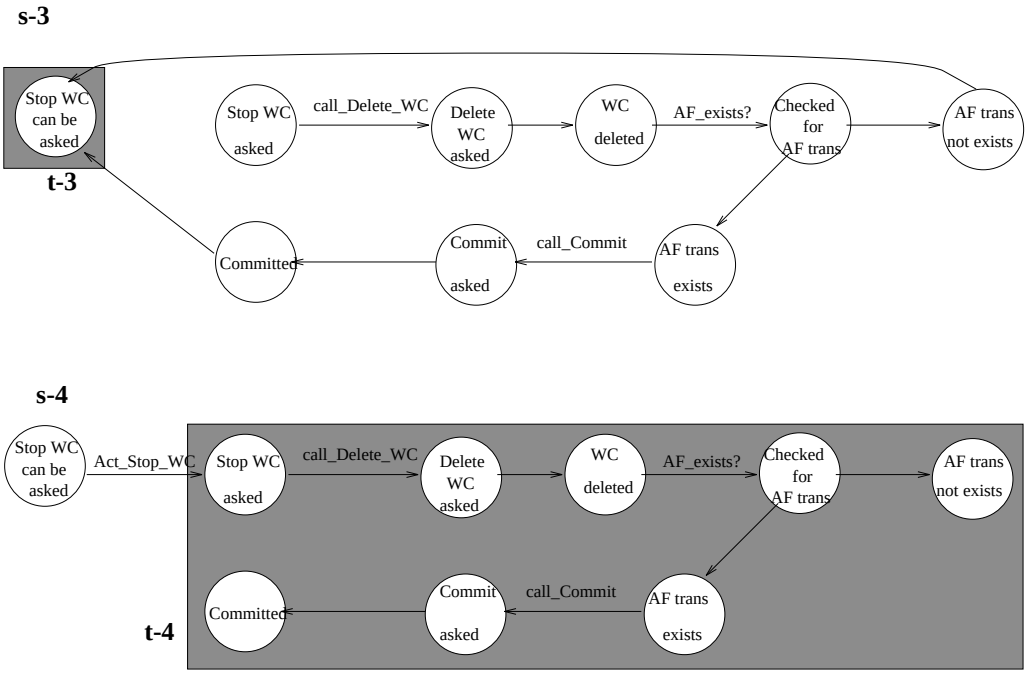
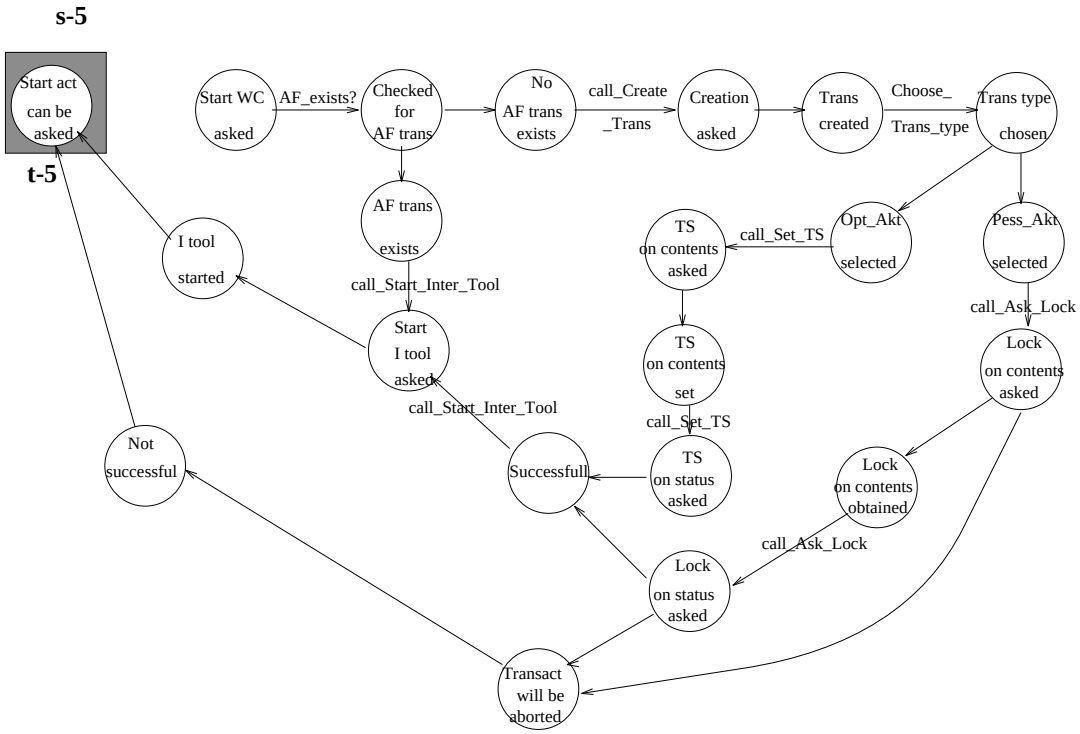


FIGURE 84.int-Stop_WC's subprocesses and traps w.r.t. Process Engine.



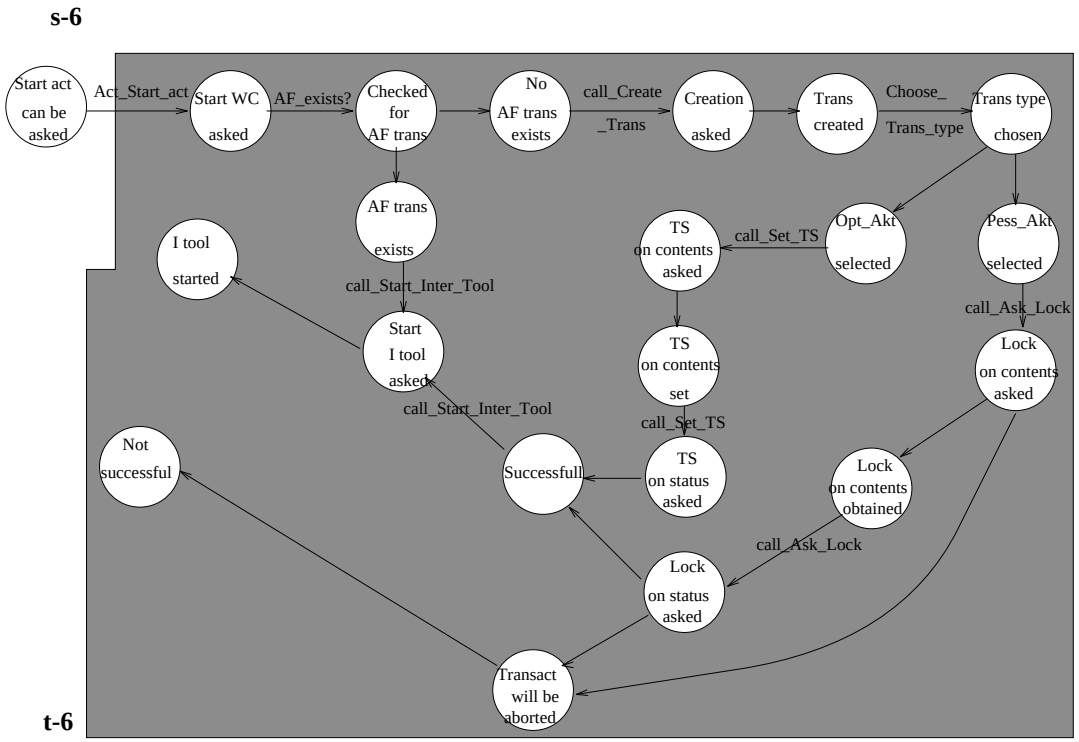
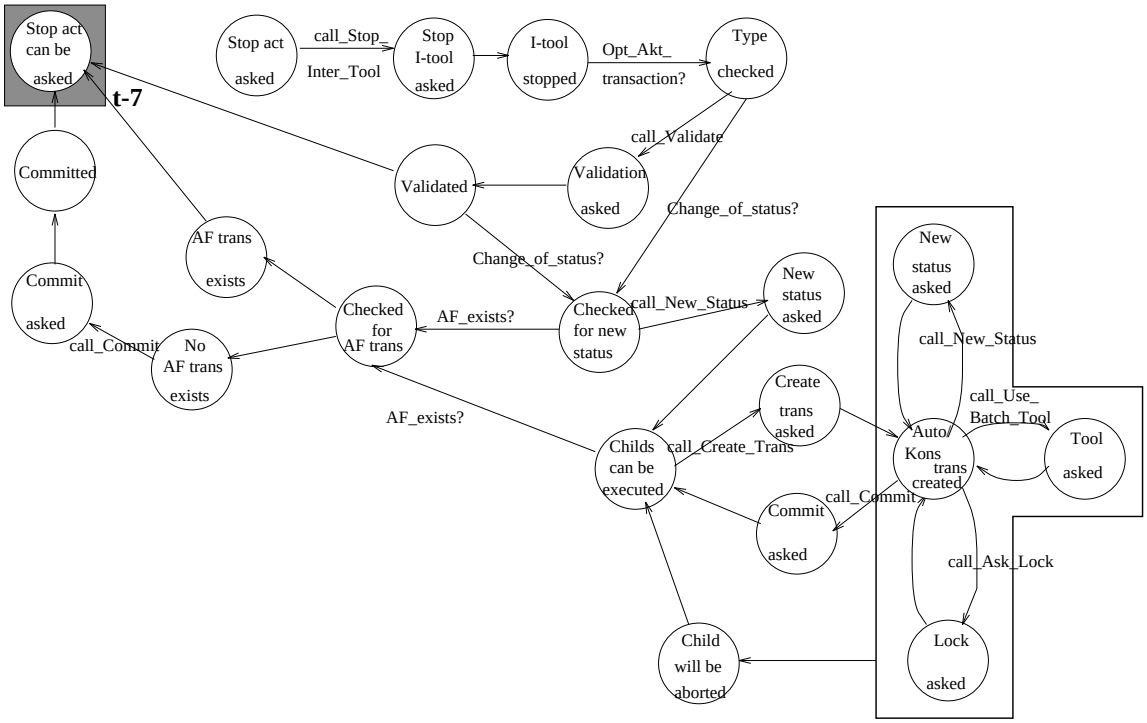


FIGURE 85.int-Start_Act's subprocesses and traps w.r.t. Process Engine.



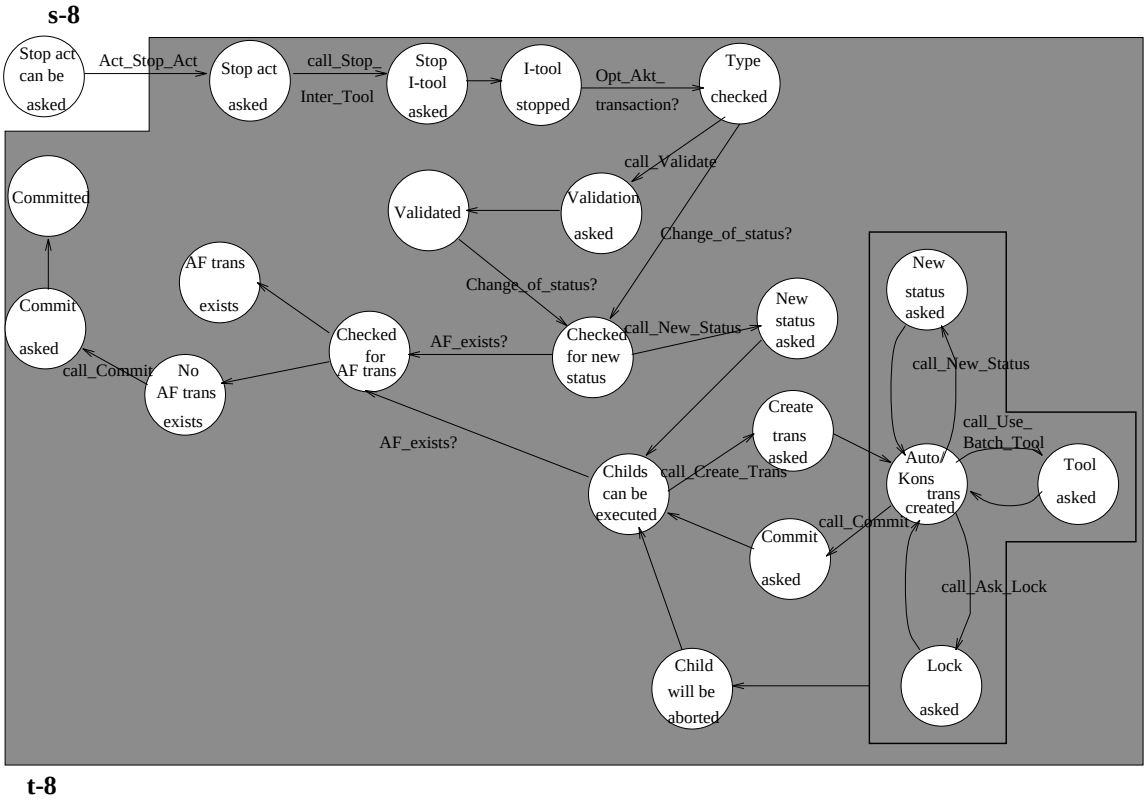
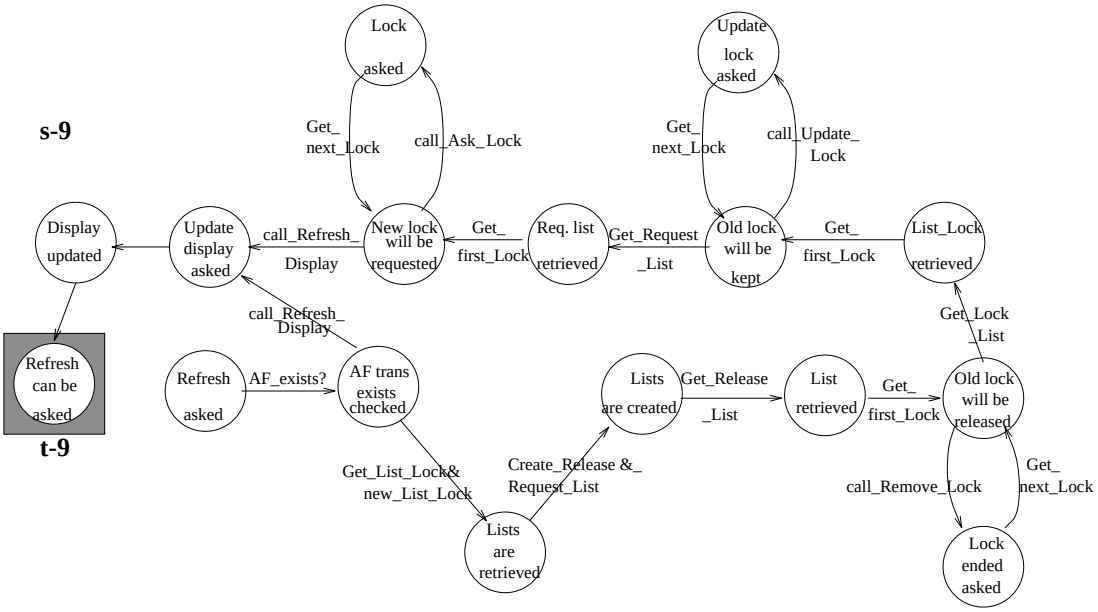


FIGURE 86.int-Stop_Act's subprocesses and traps w.r.t. Process Engine.



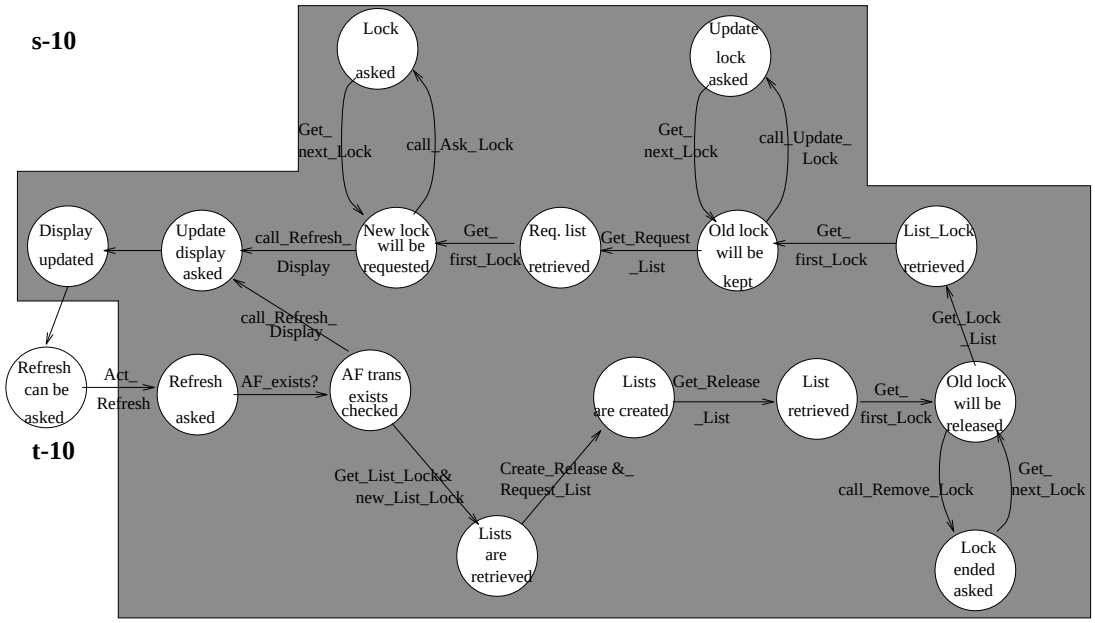


FIGURE 87.int-Refresh's subprocesses and traps w.r.t. Process Engine.

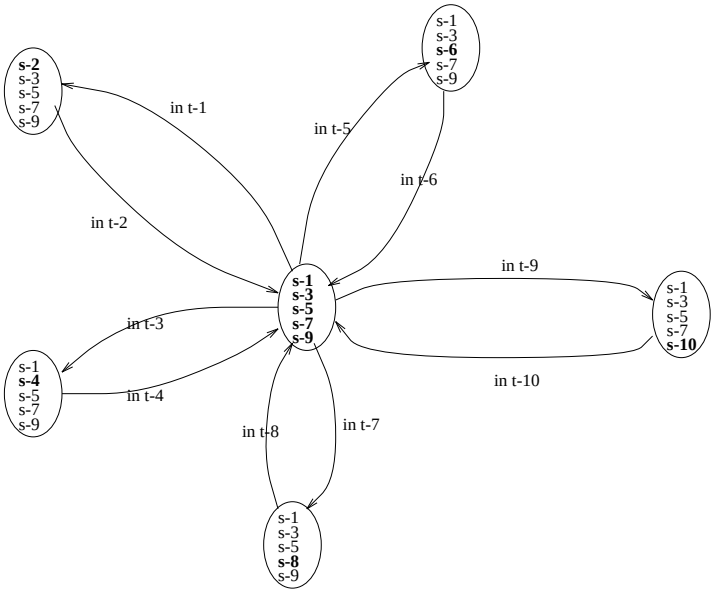


FIGURE 88. Process Engine, manager of five employees.